ON THE PROBLEM OF PARALLELIZING MANIFOLD COVERING ALGORITHMS

BY

PRATIK MALLYA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisor:

Professor Harry Dankowicz

ABSTRACT

Continuation methods are numerical algorithms used to determine the solution space of systems of nonlinear equations with associated sets of parameters. Such methods have been very successful in computing solution manifolds of dimension one. For higher dimensional manifolds, different techniques have been tried, with one method, *Henderson's Algorithm*, offering the most promise. However, the enormous size of the systems encountered in practice, along with the high dimensionality of the solution manifold, may make the method too slow for practical use.

This thesis evaluates an approach for the parallel computation of manifolds. We experiment with a few variations before deciding on an approach that proves most promising. We use the COCO toolbox, written in MATLAB, for all our experiments. In particular, we make use of MATLAB's *Parallel Computing Toolbox*, which provides the infrastructure for limited parallel processing. In the course of our work, we discuss various issues faced when computing manifolds in parallel, such as the efficient merging of manifolds and accurate estimates of performance improvement over corresponding serial methods. In the concluding chapters, we show some results that were obtained using our implementation and discuss improvements that might make the algorithm even more efficient.

To Rohini Manjunath Kamath (1942-2010)

ACKNOWLEDGMENTS

Any significant body of work cannot be done in isolation, it is the work of many people and unfortunately there is only one place to acknowledge all the people who made such a contribution. Nevertheless, I would like to take this opportunity to thank all those who have helped me in the completion of this work.

Firstly, I extend much thanks to my advisor, Professor Harry Dankowicz, for providing me with an interesting problem to work on and an environment that helped me grow intellectually. I'm glad that I had the opportunity to work under an advisor who brought out the very best in me. It has been a pleasure working and learning from him, and I consider it to be a privilege to have had ready access to his knowledge and opinions. I am especially thankful for him to have made himself available for discussion in the past few months, without which this thesis could not have possibly been completed.

I would also like to thank Professor Matthew West for the variety of discussions we had on parallel computing, and much else. Thanks to Professor Anil Hirani for sparking my interest in Numerical Analysis, an exciting field which I hope to explore further in the future.

The University of Illinois at Urbana Champaign, especially the Department of Computer Science, has been very helpful to me, easing the difficult process of settling in for a person from a different country and culture. I thank them for all their efforts to make life as a graduate student as fun and productive as possible. In addition, the Department of Mechanical Science and Engineering made it very easy and convenient for a CS graduate student to work in their department.

I would also like to thank Enthought Inc. for being supportive of my efforts to complete this work.

Finally, family and friends are the bedrock of my personality; they have been there to counsel me in bad times and to celebrate with me when times

are better. Without their support and encouragement, I would have given up long ago. As always, special thanks are due to my mother, Geeta Kamath and my father, Giridhar Mallya.

The code made available in this thesis was developed jointly by me and Professor Harry Dankowicz.

This material is based on work supported by the National Science Foundation under Grant No. 1016467.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

One of the biggest unsolved computational problems of today is the accurate and efficient determination of solutions to a general system of nonlinear equations. Such equations arise frequently in diverse areas, from engineering and medicine to economics. Perhaps an often overused example is the prediction of weather [1], but nonlinear equations have been used to describe esoteric managerial phenomena [2], as well as to predict the behavior of financial markets [3] (although recent events [4] have cast serious doubt on the reliability of such methods). Many of the most important problems in the above-mentioned areas are described by nonlinear systems of enormous sizes, having thousands of variables and equations.

A system that maps the values of input variables to output variables can be represented as a function $f : \mathbb{R}^n \to \mathbb{R}^n$. If this system violates the superposition property

$$f(\alpha x + y) = \alpha f(x) + f(y) \tag{1.1}$$

it is said to be a nonlinear system. The quadratic polynomial $f(x) = x^2$ is a simple nonlinear algebraic function.

Consider the following example problem that gives rise to a system of nonlinear differential equations. In a predator-prey system, the population of a predator can be represented as a time dependent scalar variable $x(t)$, while the population of a prey may be represented as a scalar variable $y(t)$. The equations governing the evolution of this system in time in an isolated environment are given by [5]:

$$\dot{x} = xf(x, y) \tag{1.2}$$

$$\dot{y} = yg(x, y) \tag{1.3}$$

where, $f(x, y)$ and $g(x, y)$ are any scalar-valued functions such that $\frac{df(x,y)}{dy} < 0$ and $\frac{dg(x,y)}{dx} > 0$. The Lotka-Volterra model proposes the functional expressions $f := b - py$ and $g := rx - d$, giving the system:

$$\dot{x} = x(b - py) \tag{1.4}$$

$$\dot{y} = y(rx - d) \tag{1.5}$$

This system of equations is *nonlinear*; as can be seen by the fact that the map from initial conditions to forward-time solutions does not satisfy the superposition property.

It is only the exceptional case that a nonlinear system of equations may be analyzed in terms of an explicit input-output relationship. As an example, it is not possible to obtain an explicit solution of the Lotka-Volterra model in terms of elementary functions. Instead, we must rely on numerical methods [6]. There exist many such nonlinear systems that do not have any analytical solutions and must be analyzed using numerical techniques. Most of the computer programs that are run on the biggest supercomputers of today are nonlinear numerical solvers that are used to analyze such systems [7].

Several methods have been proposed to solve nonlinear systems. When computing forward-time trajectories, the Lotka-Volterra system can be solved by using a numerical integration method. Other methods, particularly well-suited to boundary-value problems, are known as *Newton type* techniques, as they use a variation of the *Newton method* for root finding. The Newton method is an iterative method for determining the roots of a system of functions. Briefly, in the n-th iteration, we use the computed approximate solution point $\mathbf{x_n}$ to compute the next solution point $\mathbf{x_{n+1}}$ by solving the system of *linear* equations:

$$J_F(\mathbf{x_n})(\mathbf{x_{n+1}} - \mathbf{x_n}) = -F(\mathbf{x_n}) \quad F : \mathbb{R}^n \to \mathbb{R}^n \tag{1.6}$$

For scalar problems, this requires the existence of the first derivative on some neighborhood of the solution point and through the solution manifold; for a function $F : \mathbb{R}^n \to \mathbb{R}^n$, this condition reduces to ensuring that the Jacobian (J_F) function is nonsingular and well-conditioned near the solution manifold.

In many situations we do not require the explicit solutions to the original

problem; we might be interested in a subset of solutions that also satisfy an additional constraint. As an example, we might desire to explore the roots of the function

$$f(\mathbf{x}) = \sin(x_1) + 3e^x \cos(x_2) \tag{1.7}$$

over the curve defined by the constraint

$$x_1^2 + x_2^2 = 1, \tag{1.8}$$

In this situation, it doesn't make much sense to determine all possible solutions of the function in Eq. (1.7) (as most of the solution points won't satisfy the constraint in Eq. (1.8)). We therefore look for other techniques of achieving our goal of analyzing the behavior of our system.

It turns out that there do exist such methods; we discuss a particular technique called *Numerical Continuation*, which is described in detail in section 2.1. We are interested in numerical continuation in higher dimensions, preferably for any dimension n, as such an approach may allow us to observe how the solution manifold depends on a multitude of system parameters. To this end, we explore an implementation of the algorithm from [8] in a MATLAB based continuation toolbox [9]. Our original algorithm development is motivated by the observation that the existing implementation scales poorly with problem and manifold dimension.

At the present time, microprocessor speeds for a single processor have peaked; chips cannot run any faster without requiring some serious dedicated cooling systems. Instead, microchip manufactures have turned to providing more computing cores on the same chip. Thus, the only avenue for speeding up existing computational algorithms is to use some sort of parallelism or concurrency, and thus have more processors work on the same problem. This strategy for obtaining better performance from existing algorithms has been espoused since the very beginning of the computer age; it is no surprise that this thesis looks upon parallelization as the most promising route to our end goal of lowering the computation time. The specific strategies that we employed, and both our failures and successes in our attempt to attain this goal, form the core of this thesis.

The rest of this thesis is organized as follows: Chapter 2 consists of a brief introduction to numerical continuation, computational geometry and

parallel programming. Chapter 3 gives an introduction to a MATLAB based continuation toolbox and examples of how it may be used. In Chapter 4, we discuss the original contribution of this thesis: a method for parallel covering of an implicitly-defined manifold. Finally, Chapter 5 summarizes our contributions and observations and makes suggestions for future work.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we shall give an introduction to the theoretical ideas most relevant to our goal of computing manifolds in parallel. This discussion will aid in understanding the concepts that will be used extensively throughout our work.

We begin in Section 2.1, where we give an introductory description of continuation. Section 2.2 describes some concepts of computational geometry used frequently in the remainder of the text. Indeed, we make immediate use of the concepts from the previous two sections in Section 2.3, where we discuss numerical continuation in higher dimensions. Finally, in Section 2.4, we take a short tour of the enormous work in parallel and concurrent computation that has been the inspiration for some of the techniques used in our original development.

2.1 Numerical Continuation

Continuation algorithms refer to a class of numerical methods that are used to find families of solutions to a system of nonlinear equations. Such methods are particularly useful when we desire to determine the dependence of the behavior of the system on additional parameters. In the following sections, we shall go into some detail about the techniques used in numerical continuation, especially for continuation in higher dimensions.

If we consider a system of nonlinear equations that is described by:

$$F(u, \lambda) = 0, \qquad F : \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^n \tag{2.1}$$

then continuation algorithms may be used to construct the solution manifolds of this system, i.e., the values of the input variables $u \in \mathbb{R}^n$ that satisfy the equations for given values of the system parameters $\lambda \in \mathbb{R}^k$.

2.1.1 Classes Of Continuation Methods

Continuation methods can be broadly classified into two major categories: *Simplicial Continuation* and *Pseudo Arc-Length Continuation* (PSALC) methods [10].

Simplicial Continuation assumes the existence of a reference simplicial decomposition of the domain $\mathbb{R}^n$. A simplicial decomposition is simply a way to cover a domain with simplices which do not overlap; in other words, it is a discretization of the domain using simplices (which are generalizations of triangles to higher dimensions; we explain these concepts in greater detail in the next section). This method tries to determine all the simplices of the reference decomposition which contain the solution manifold. Hence, the manifold M is represented by a set $L(M)$ of the simplices which contain it.

To see how this is possible, consider that, if the solution manifold is k-dimensional, then k additional constraints are required to uniquely identify a particular point on the solution manifold. In other words, a point on this manifold can be said to be an intersection between it and another $(n-k)$-dimensional manifold. Hence, if a solution manifold intersects an n dimensional simplex, it will do so at a point on a $(n-k)$-dimensional face ρ. We can now choose the set of simplices that contain ρ in their interior; these are the simplices that we can test for the next continuation point. Once the simplex corresponding to the next solution point is determined, it can be added to the growing manifold M by simply adding it to the set $L(M)$ (the simplex is said to be *merged* into the manifold). We are not required to check for overlap between the added simplex and the simplices in $L(M)$ because all the simplices are obtained from a reference simplicial decomposition of $\mathbb{R}^n$. Since the reference decomposition is a simplicial decomposition, every part of the domain is covered by one and only one simplex (the decomposition is said to be *compatible*). The advantage of such a simplicial decomposition of $\mathbb{R}^n$ is principally to obtain an easy way of merging new simplices into the manifold representation $L(M)$.

We glossed over the process of determining whether the simplex contains the solution manifold. We can think of the function $F(\mathbf{v}) : \mathbb{R}^{n+k} \rightarrow \mathbb{R}^n$ in Eq. (2.1) as mapping the simplex from one space to another (this might be clearer if the reader tries to visualize each vertex $\mathbf{v_i}$ of the simplex being transformed into a vertex $F(\mathbf{v_i})$; the new vertices then define a new simplex).

If the solution manifold passes through this simplex, the mapped simplex will contain the origin (as all points on the solution manifold evaluate to zero under $F(\mathbf{v})$). Hence, the check for manifold intersection reduces to a check for the origin in the interior of the mapped simplex.

A concrete example of this process is given by contour plotting in 2D. Suppose we are trying to plot the contours of the equation

$$F(\mathbf{x}) = 0, \qquad F : \mathbb{R}^2 \to \mathbb{R} \tag{2.2}$$

We obtain a simplicial decomposition of the plane by covering it with triangles. We determine which triangle the initial point belongs to and then check the neighbors of this triangle. This check is done by evaluating the value of the function $F(\mathbf{x})$ at each of the vertices of the triangles; if there's a change in sign along an edge, it means that $F(\mathbf{x}) = 0$ somewhere on it. This point in then determined by an interpolation between the values of $F(\mathbf{x})$ at the endpoints of the edge.

In *Pseudo Arc-Length Continuation*, we construct the solution manifold by making successive local extensions on neighborhoods of regular solution points. A *regular point* of the solution set is a point u_0 where the Jacobian $F_u(u, \lambda)$ has full rank (here, we are referencing Eq. (2.1)). The Implicit Function Theorem (IFT) guarantees the existence of a *locally unique* k-dimensional manifold of regular points in some small neighborhood of u_0 [11]. Starting with the regular point, we attempt to compute the locally unique solution manifold by finding nearby regular solution points, and so on and so forth. If we encounter a singular point, the IFT no longer applies: e.g., uniqueness of the manifold may no longer hold, and we may have to choose amongst the many manifolds intersecting at this point. In practical implementations, this process consists of computing *charts* (each of which represents the solution manifold using a regular point $\tilde{u}$ and a small neighborhood around it). A collection of charts is known as an *atlas*; such an atlas is said to *cover* some portion of the manifold. As discussed in later chapters, *atlas algorithms* use a chart to compute its neighboring charts, *merge* the newly computed charts into the growing atlas, and terminate the computation after a sufficient cover has been achieved.

The major difference between these two methods is that the former computes a cover of the manifold in terms of a *simplicial decomposition* of the

domain $\mathbb{R}^n$ whereas the latter computes the cover in terms of a simplicial decomposition of the solution manifold, $\mathbb{R}^k, k < n$. While it might appear that obtaining a simplicial decomposition of a manifold of lower dimension is easier than that of the whole domain, there are other properties that might convince us otherwise. For instance, the merge process is harder in PSALC methods; the new simplices being merged into the growing manifold have to be guaranteed to be *compatible* with each other (i.e. a suitably defined intersection of two simplices is either another simplex in the complex, or it is empty); whereas in simplicial continuation, the simplices used to represent the manifold are chosen from a reference decomposition which is compatible (by construction).

The history of development of continuation algorithms testifies to the above factors. PSALC methods are most successful in 1D as the merge process is easy; it consists of simply adding or dropping intervals along a curve. In higher dimensions, progress has been slower on both methods, but obtaining simplicial decompositions of $\mathbb{R}^n$ for high values of n is a hard problem. Instead of generating a new simplicial decompostion for every problem, we might instead choose among some reference decompositions, some of which are enumerated in [12]. The problem with using a reference decomposition for every problem is that regions where the solution manifold has very high curvature might be insufficiently covered. For these reasons, PSALC methods have been more popular for continuation in higher dimensions.

We now make an important observation regarding the solution manifolds obtained using regular points. The IFT guarantees the existence of a locally unique solution manifold near regular points. This property, while helping us locate neighboring points on the solution manifold, has another significant implication: a regular point cannot be a boundary point. That is, a branch cannot just terminate; it has to either self-intersect, approach infinity or be bounded by a set of singular solutions (or a combination of the above). This means that the solution space appears as a set of k-dimensional manifolds that are connected to each other at lower dimensional singular spaces [13].

In the next section, we give a brief description of computational geometry, where we will introduce mathematical concepts that will be used to continue our discussion on numerical continuation in Section 2.3.

2.2 Computational Geometry

Computational geometry is concerned with the problem of devising algorithms and data structures for the efficient solution to geometric problems. Efficiency in this case refers to polynomial running time in the size of the input. Geometric problems are those problems which require the use of geometry, both for their description and their solution. For example, if we consider the problem of finding all the points in area A, we use the intuitive geometric concept of *point* and *area*, and hence this is a geometric problem. We represent these concepts in a manner that can be efficiently and accurately understood by a computer. In the above simple example, we may choose to represent points as tuples and areas as being marked by an enclosing polygon. Such a polygon may be represented by a list of points.

This representation is then manipulated by a computer to determine the solution. The entire process can be termed as *solving a geometric problem.* Note that once we have a way to translate geometric primitives into computer language, we are then concerned more about the geometric algorithms than the actual implementation of the primitives.

It is quite evident that geometric problems are all around us. For the sake of concreteness, we mention specific examples: computer graphics, industrial robotics, computer vision, CAD (Computer Aided Design), etc. A common theme that runs through these applications is that these are essentially ways to efficiently represent a 3D environment in a computer. Since our 3D environment is described by geometry, the ideas of computational geometry form a natural fit.

The study of computational geometry began in the 1970's, when CAD applications that required computer graphics, inspired researchers to come up with better and more efficient solutions to work on the computers of the time [14]. The field of computer graphics is almost entirely based on the foundation of computational geometry. The heavy requirement of modern animations, as well as the huge demand for better graphics in consumer devices has also been one of the prime motivations for research in computational geometry.

Our interests lie in using the best suitable data structures to represent *manifolds* and the process of covering these manifolds. A manifold of dimension n is a topological space that near each point resembles n-dimensional

Euclidean space. In other words, it is a generalization of the concept of a surface to higher dimensions. Indeed, a surface is considered to be a manifold of two dimensions. We have seen in Section 2.1 that the set of solutions of a nonlinear system could be described in terms of a solution manifold; this is the prime reason for our interest in this topic. The incremental process by which the manifold is determined is known as *covering* of the manifold.

Since we live in a 3D world, most extant work on covering manifolds has focused on covering surfaces. In particular, surface (re)construction has dealt with precisely this problem, motivated mainly by applications in computer graphics that require a faithful reconstruction of real world objects. Visualizations of abstract data also helps us to easily spot patterns in the data (which is difficult to do with the raw data). For example, it is much easier to spot the trend in the price of a company's stock when its plotted as a line plot instead of being enumerated in a table.

In many scientific applications, we are required to visualize an *isosurface* of the given data. An isosurface is just a surface consisting of all the points that have a particular function value. The data in these cases is usually just a collection of points, which can be regarded as an n dimensional scalar field. The *Marching Cubes* algorithm is perhaps the best known surface reconstruction algorithm [15] for obtaining an isosurface from such scalar fields. The algorithm proceeds through the scalar field by considering eight neighboring vertices at a time (which form the vertices of a cube). The value of the scalar field is computed at these eight points, and the value of the isosurface is then subtracted from each point; this value is then thresholded to give one if positive, and zero if negative. Thus, we now have an 8-bit vector, which indexes into a precomputed array of $2^8 = 256$ polygons, which can be thought of as a basis set. Once such a polygon is chosen, it is placed in the hypothetical cube, with the values of the vertices being used to determine the placement of the polygon (The astute reader might have realized that this method resembles the simplicial continuation discussed earlier).

This method is widely employed by current 3D modeling software (such as the Visualization ToolKit, VTK) [16]. The availability of better sensors (such as the Kinect®[1]) has resulted in the generation of tremendous amounts of point-cloud based data, and the resulting demand for reconstructing the

[1]Registered trademark of Microsoft Corporation

surface (which was the source of these points) has served as a strong impetus for developing more efficient methods [17, 18].

2.2.1 Geometrical Data Structure For Higher Dimensional Manifold Covering

The discussion above provides a brief introduction to the kind of research that has targeted manifold covering explicitly. We proceed to discuss the data structures that are used by algorithms for covering manifolds of high dimensions. As discussed earlier, our choice of data structures is influenced chiefly by their ability to both efficiently and faithfully represent the manifold, and by their support for operations such as merging. We will discuss four such concepts: the Delaunay triangulation, its dual, the Voronoi diagram, a variation of the Voronoi diagram known as the Laguerre-Voronoi diagram, and another triangulation known as the *Coxeter-Freudenthal-Kuhn* triangulation. Most of the material in this section has been sourced from [8].

Given a set of points u in $\mathbb{R}^n$, the *Laguerre-Voronoi Diagram* [19] of the points is a decomposition of $\mathbb{R}^n$ into non-overlapping regions, each associated with a particular point. Each point u_i is assigned a weight R_i and the boundary between neighboring regions is determined by the equation:

$$||u - u_i||^2 - R_i^2 = ||u - u_j||^2 - R_j^2 \tag{2.3}$$

Here, any suitable norm may be used; the selection of different norms gives different kinds of boundaries between the regions. In case we use the 2-norm, the above equation can be simplified to:

$$2(u_j - u_j) \cdot u = R_i^2 - R_j^2 + |u_i|^2 + |u_j|^2 \tag{2.4}$$

which describes a plane orthogonal to the line connecting the centers of the two cells. If we think about the point u_i as being the center of a spherical surface in $\mathbb{R}^n$ with radius R_i, we see that the intersection between neighboring spheres is coincident with the intersection of each surface by a common hyperplane. We will make much use of this fact later when we use spheres to construct the solution manifold, and the merge process consists of removing regions of overlap between neighboring spherical surfaces by intersections

with suitably defined planes. When $R_i = R_j, \forall i \neq j$, Eq. (2.3) reduces to:

$$||u - u_i||^2 = ||u - u_j||^2 \tag{2.5}$$

which is the defining equation of a *Voronoi* diagram. We see that a Voronoi diagram is a more specific version of the Laguerre-Voronoi diagram. The interior region of a point u_i in the corresponding decomposition is said to be closer to u_i than to any other point; here closeness is defined in terms of the norm used in Eq. (2.5).

We next discuss geometrical *triangulation* techniques. Triangulation is the process of discretization of a manifold with nodal points (on the manifold) and edges (not typically on the manifold) connecting these nodal points. It is easy to see why triangulations are important when we are attempting to solve geometric problems on a computer: they represent a kind of discretization of the domain; and to solve continuous problems with a computer requires such a representation. Hence, the discretization is crucial to the correctness of the algorithm; a poor representation means that interesting parts of the domain are missed. (This is not unlike the *Nyquist theorem* in signal processing, which essentially says that detection of high frequencies requires smaller discretizations.) Another often overlooked application of triangulations is in proving theorems in computational geometry, although in such cases, the existence of a triangulation is often simply assumed (i.e., the problem is not *how* to get the triangulation, but what to do with it once it has been obtained) [20].

We discuss here two kinds of triangulations that are used extensively in continuation algorithms for triangulation of the solution manifold. They are the *Delaunay* triangulation [14] and the *Coxeter-Freudenthal-Kuhn* triangulation [21].

In $\mathbb{R}^2$, the *Delaunay triangulation* can be obtained from the Voronoi diagram discussed earlier; it is the dual graph of the Voronoi diagram. The *dual graph* of a planar graph is the graph obtained by using a vertex for each face in the original graph, and an edge between two vertices if they correspond to neighboring faces in the original graph. It turns out that Delaunay triangulations maximize the smallest angles in the resulting triangles; this property is extremely desirable of a triangulation, especially when used for computing functions on the manifold [14].

The Coxeter-Freudenthal-Kuhn triangulation is a robust method of extending an existing simplicial decomposition. If we have a simplicial decomposition of only a part of the domain, it is easy to see that the boundaries of the structure are made of cells of dimension n. These cells have *boundary faces* (which are simplices of dimension $n-1$) that have the interior of the structure on one side of the face, and the exterior of the structure on the other. This fact can be used to create a simplex across such a face by reflecting the simplex across the face. This process of moving across a face to the adjacent simplex is called *pivoting*, and is used to extend the simplices in the direction of interest.

2.3 Higher Dimensional Continuation

We shall now use the concepts discussed in the previous two sections to describe the process of covering higher dimensional manifolds.

2.3.1 Mathematical Tools

The mathematical structure that we use to represent the manifolds is called a *complex*, which is a generalization of a mesh to higher dimensions. Data structures to represent geometrical and topological algorithms are, in general, hard to design [14]. It isn't hard to see why: representing geometrical objects such as points is easy, and constructs such as lines and planes might possibly be described with the help of stored equations. But, what about a general surface? Surfaces are continuous geometrical objects, and to represent them by a computer, we have to use some kind of discretization. It turns out that meshes are the best way to represent general surfaces, and simplicial and cell complexes (explained below) are the best way to mathematically define these meshes. It should be noted that these representations devolve into simpler ones when computing manifolds of lower dimensions; curves can be easily represented by a set of points, and their boundaries by their start and end point. We try to use innovative (but perhaps slightly unintuitive) data structures in order to deal with manifolds of higher dimensions, and as shown in [8], these data structures are capable of faithfully representing manifolds of any dimension.

We consider a *cell complex* of dimension k, embedded in $\mathbb{R}^n$, to be a set S of convex polyhedra (called *cells*) of dimension 0 to k, which satisfy two conditions

- Every face of a cell of dimension $2 \leq p \leq k$ is a cell of dimension $p-1$ in S;
- If R_1 and R_2 are two cells in S, then their intersection is either empty, or a face in S common to both.

Here, the 0-cells may be identified with points on an n-dimensional embedding space.

These properties impose structure on the set of cells represented by the cell complex; in particular, when merging two cell complexes (e.g., when merging two manifolds, discussed later) the algorithm must ensure that the resulting structure is also a cell complex.

The convex polyhedra that constitute the cell complex have an interesting structure: they may be specified simply by specifying the hyperplanes that contain the faces. These hyperplanes represent the boundaries of the half-spaces enclosed by the polyhedra. This choice of representation is effective for one operation that interests us greatly: that of subtracting a half-space from the polyhedron. It is evident that this operation requires only the addition of an additional constraint specifying the hyperplane that forms the new boundary.

A k-dimensional *manifold* is a set of one-to-one, continuous maps on k-dimensional neighborhoods of the origin (called *charts*) that are isomorphic to the k-dimensional unit ball $B = \{\mathbf{x} || \mathbf{x}| < 1\}$, along with adjacency relations indicating which charts 'overlap'. Since charts are structures used to model a continuous structure, they must agree on some common non-empty subregion that establishes the correspondence from one chart to another (there is a surjective mapping between the common regions). The collection of all charts is called an *atlas* (analogous to those used for maritime navigation) for the manifold. Finally, the relation between a manifold and a cell complex is that we may represent each chart by a k-cell, and describe the overlap between neighboring charts in terms of a $(k-1)$-cell in the complex.

For the purpose of continuation, we need a way of representing charts that are on the boundary of the atlas; for this provides us with more regular

points to start continuation from. Hence, we keep track of the set of charts on the boundary of the atlas, the *boundary charts*, typically by enumerating these in a list. Adding external aspects to any mathematical structure has its drawbacks, however: while it allows us to better describe manifold based methods, these methods now have to explicitly keep track of the boundary charts, updating them to be consistent if any operations are done on the manifold.

This concludes our discussion of the geometrical and topological tools that we shall use in the next section.

2.3.2 Henderson's Algorithm for Higher Dimensional Continuation

There exist many algorithms for higher dimensional continuation, five of these are listed in [8]. In this section, we discuss one of these methods, the Henderson algorithm for higher dimensional continuation.

In this algorithm, individual charts provide a local cover of the solution manifold M described in terms of base points on M and geometric structures in the corresponding tangent spaces. Specifically, a chart based at a point u_i on M is represented by:

- an orthonormal basis T_i for the tangent space;
- a sphere of radius R_i in the tangent space; and
- a polyhedral k-cell P_i.

The algorithm is initialized by constructing a basis T_0 for the tangent space of the initial base point u_0 and constructing a sphere in this tangent space with u_0 as its center. A cube P_0 is then created which is slightly larger than the sphere, so that its vertices lie outside the sphere. At any stage of the algorithm, a boundary point can be found by locating a base point u whose associated chart has at least one vertex v outside the sphere. The intersection of the line uv with the sphere then gives us a boundary point w.

Since the IFT guarantees the presence of a locally unique manifold passing through a regular point, there must exist such a manifold near w. We determine a basis T for its tangent space by observing that finding the tangent space corresponds to finding a nullspace of the Jacobian J_F of the function F

(defined in Eq. (2.1)). Therefore, we have the following system of equations for T:

$$J_F(u) \cdot T = 0 \tag{2.6}$$

$$T^{\mathrm{T}} \cdot T = I \tag{2.7}$$

Now, the neighborhood of w on M that is covered by the corresponding chart is just the projection of the corresponding sphere onto M. Again, P_w is initialized to a cube centered on w which is slightly larger than the sphere.

Merging the new chart into the existing atlas corresponds to finding neighborhood spheres and then ensuring that their respective polyhedra do not overlap when projected onto the corresponding tangent spaces. Recall that the intersection of two spheres corresponds to the intersection of each sphere with a common hyperplane. This plane can then be used to chop the respective polyhedra. This ensures that no vertex of one polyhedron lies inside its neighbor. Essentially, this operation makes the polyhedra compatible.

Note that before comparing two neighboring points, we must project their sphere and polyhedra into the tangent space of the neighbor. For instance, if u_i and u_j are neighboring points, then to check whether they overlap we take the following steps:

1. Project the sphere (that corresponds to u_i) to the tangent space of u_j.
2. Check for overlap between the sphere of u_j and this projected sphere of u_i.
3. If such an overlap does exist, find the hyperplane of intersection and use it to chop P_j to get P_j'.
4. Repeat the same process (but with the original sphere and polyhedron, P_j) using u_j to get P_i'.
5. Replace P_i and P_j with P_i' and P_j' respectively.

2.4 Parallel Algorithms

In this section, we go into some detail on the vast topic of algorithms for parallel computation. Several volumes have been written on this topic, which has been an area of active research ever since the invention of modern computers. We shall be very brief, and try to limit our discussion to those topics that we found relevant to our algorithm development.

2.4.1 Introduction to Parallel Computation

How do we complete task A quicker? The immediate solution that comes to mind is to have more resources devoted to it. In the case of solving a task with computers, we have the constraint that a single processor can only run so fast. Hence, using multiple processors to complete the given task was the natural answer in cases where faster completion times were desired. Whether this *parallel* solution (which takes time T_p) is better than the serial algorithm (which takes time T_s) is measured with the help of metrics such as *Speedup*

$$S = \frac{T_s}{T_p} \tag{2.8}$$

and *Efficiency*

$$E = \frac{S}{N} \tag{2.9}$$

where N is the number of processors used [22].

In most cases, the parallel algorithm for a given task involves some form of communication between the processes to coordinate the completion of the task. This time overhead is termed as the *communication overhead* and one of the primary objectives of efficient parallel algorithms is to minimize it.

Dividing a serial task across parallel processes can be done in broadly two ways:

- Execute the same algorithm on different parts of the data (known as *Data Parallelism*).
- Execute different parts of the algorithm on the same or different data (known as *Task Parallelism*).

Most graphics programs are highly data parallel. This gave rise to specialized

processors (known as *Graphics Processing Units* or GPU) which solve very highly data parallel programs by devoting more chip space to ALU's (Arithmetic and Logic Unit) at the cost of cache memory found in conventional CPU's. The trade off is that GPU's cannot be used for more general computation tasks; this is the reason why modern High Performance Computing systems often consist of both CPU's and GPU's. The overall control (and general computation) is handled by the CPU but data-parallel parts of the application are delegated to the GPU. For instance, a computer game might consist of an interactive user interface, which is handled by the CPU; but the actual process of computing which images should display on the computer screen is a computationally intensive, yet repetitive task, which is offset to a GPU.

Many software libraries have been developed to make it easier to develop software that can harness the computing power of multiple processors. The most popular of these libraries include Pthreads, MPI and OpenMP for CPU and CUDA®[2] for GPU. These libraries provide primitives that allow efficient communication and coordination between different processes (that may run on the same or different machines) in solving a large problem. For example, the MPI primitive `MPI_Gather` is used to make the program wait for all the parallel processes to finish a certain task.

Industrial research has been more focused on parallelization of entire applications. This field uses commodity hardware and software tools to tackle problems spanning very large datasets (on the order of several terabytes) and is generally known as *Distributed Computing*. One notable publicly available tool is Apache Hadoop, which is a set of libraries that allows for both storage and processing of large datasets on a variety of hardware. It is an implementation of the *Map-Reduce* [23] paradigm, which is explained below.

When we are dealing with datasets spanning several terabytes, it would not be practical to perform any analysis without some form of distributed computing.. Hence, some method is required to distribute (*map*) the data over a large number of processors and then have an efficient way to combine (*reduce*) the results. In the map-reduce paradigm, input data is divided into chunks and tagged with many fields; these chunks are small enough to be individually processed efficiently by a single processor. Once the results are

[2]Registered trademark of NVIDIA corporation

computed, a global process then scans across these tags, selecting those that are required in the further analysis.

2.4.2 Parallel Algorithms for Surface Covering

Surface reconstruction was mentioned previously as a field which has generated many ideas that can be used for manifold covering. Such algorithms are becoming even more important with the availability of extremely large datasets. These datasets may arise from various sensors such as LIDAR (*Light Radar*), which uses laser beams to accurately measure distances [24]. These sensors usually output a series of point-based data, which serves as a representation of the surface being probed. To aggregate these points and recover the surface is the principal aim of most surface reconstruction algorithms. When these datasets become too large to be analyzed by a single machine, the task must somehow be distributed across several machines.

We mentioned the *Marching Cubes*(MC) method earlier; much work has been done in an effort to speed up MC on large datasets by intelligent computation and parallelization. A recent work by Akinci et al. [25] uses MC independently on every particle to determine the surface of a particle based fluid. They solve the difficult problem of obtaining neighborhood information by using a *scatter-gather* primitive (very similar to *map-reduce*). A very simple illustration of this concept is: to 'paint', in parallel, each vertex by a marker indicating its distance away from every neighbor. The *scatter* primitive consists of creating this painting of each particle. In the *gather* step, this data is then used to get aggregate points.

2.4.3 Software Tools For Parallel Computation

Parallel computing software can be classified based on the assumptions made about the underlying memory model. If the parallel processes communicate by access to shared memory variables, it is known as a *shared memory architecture*. OpenMP and Pthreads are the 2 most popular libraries for shared memory parallel processing. If the processes communicate by passing messages, it is known as a *distributed memory architecture*. MPI is the most popular library for this kind of parallel processing [26]. All the above li-

braries are written in C, but bindings are available for other languages as well.

Recently, libraries for accessing GPU's for general computation have become very popular. These are used for offloading highly data parallel parts of a code to a GPU for fast processing. Note that the memory model used here is the shared memory model. The most popular library for such kind of parallel processing is CUDA, developed by NVIDIA [27]. The configuration that has found most success has been a mix of GPU with CPU in the servers that are used for heavy-duty computation.

2.4.4 Parallel Computation in MATLAB

Mathworks provides a Parallel Computing Toolbox [28] for MATLAB®[3]. This toolbox provides different paradigms for parallel programming: although primarily based on a message passing model, it also provides tools for accessing GPU's. Functions are also available for automatic parallelization of loops.

The most powerful feature provided is to launch a number of processes, all running the same MATLAB code. These processes can then be programmed to communicate and coordinate to complete a task. Currently, MATLAB supports a maximum of 12 processes with a single parallel computing toolbox license. In this work, we make use of this feature of MATLAB to speedup computation. In the remaining part of this section, we go into some detail on how to use this toolbox.

Amongst all the features offered by the Parallel Computing toolbox, the **`spmd`** construct is the most general one. It provides a distributed memory type of parallel computing framework. Essentially, a number of parallel processes, each running MATLAB, can be started at the same time. Therefore, we have n different instances of MATLAB which can communicate with each other via messages. However, these processes do not have the ability to run graphical tasks; hence the workers cannot be used for plotting (they can output data to `stdout` and write to files). Instead, the data must first be passed back to the main MATLAB process; and can then be plotted. Every parallel process has a unique index associated with it, which is stored in its `labindex` property. All workers are identical except for this one property. Workers communicate

[3]Registered trademark of Mathworks Inc.

between each other by sending data via the non-blocking `labSend` function, and block on calls to `labReceive`, waiting for a message from any, or a particular worker. Blocking can be avoided by using the convenient `labProbe` function that checks if there is any message to be received from any or a particular worker. The message sent by `labSend` is stored in a queue on a receiver's buffer. Our impression is that the **`spmd`** construct is intended for rudimentary coarse-grained parallel computation because each of the threads is 'heavy' (i.e. they each run a full MATLAB kernel). Since the toolbox is closed-source, we can do little beyond make intelligent guesses on this point.

Although we said that the parallel processes adhere to a distributed memory model, there is one class of shared memory objects called *composite objects*. Such an object is created on the client and has an entry for each worker; from here the client can access the value of the object on each worker.

Although limited in scope, the MATLAB's parallel computing toolbox offers us a sandboxed environment for parallel computing. Its biggest advantage is its ready integration with existing MATLAB tools and the same ease-of-use that made MATLAB itself popular. We exploited these features to experiment with parallel numerical algorithms, and we found it extremely useful at the prototype stage. For deployment in special cases, the MATLAB code can be used as a template for writing the same algorithm in a lower level language like C, which is closer to the machine and can hence achieve a more optimal performance.

CHAPTER 3

COCO: AN EXTENSIBLE PACKAGE FOR CONSTRUCTING AND SOLVING CONTINUATION PROBLEMS

The Computational Continuation Core (COCO) is a MATLAB based continuation platform which can be used to formulate and solve continuation problems [9].

We begin the discussion in this chapter by illustrating the use of COCO to solve an example problem. In Section 3.2, we see how a problem is formulated using the syntax of COCO. In Section 3.3, we provide provide a more detailed description of the structure of the continuation platform and give an overview of its API. Finally, in Section 3.4, we comment on the actual implementation of COCO.

3.1 Motivating Example

Let us consider the Ordinary Differential Equation (ODE):

$$\dot{x} = \phi(x, \lambda, \kappa) = \kappa - x^2 + x^4 + x\lambda \tag{3.1}$$

The values of x for which

$$\phi(x, \lambda, \kappa) = 0 \tag{3.2}$$

correspond to equilibrium points of the ODE, for certain chosen values of λ and κ. The set of equilibrium points forms a two-dimensional manifold in $\mathbb{R}^3$.

Let the function $\psi(x, \lambda, \kappa)$ be defined as:

$$\psi(x, \lambda, \kappa) := \partial_x \phi(x, \lambda, \kappa) = -2x + 4x^3 + \lambda \tag{3.3}$$

Then the solution manifold of the system:

$$\phi(x, \lambda, \kappa) = 0 \tag{3.4a}$$

$$\psi(x, \lambda, \kappa) = 0 \tag{3.4b}$$

consists of a set of non-hyperbolic points that forms a submanifold of dimension 1.

Suppose that we are given an approximate equilibrium point

$$P_1 = (x_1, \lambda_1, \kappa_1) \tag{3.5}$$

and that $\psi(P_1) \neq 0$. We wish to locate and continue along a submanifold of non-hyperbolic equilibria using the following strategy. Starting with the initial solution guess P_1, we apply continuation to the equation:

$$\phi(x, \lambda, \kappa_1) = 0 \tag{3.6}$$

Since we now have one equation in two variables, the solution will be a 1D branch that lies on the solution manifold of Eq. (3.2). If $\psi(x, \lambda, \kappa_1)$ changes sign along this branch, then there must exist a point on this branch where $\psi(x, \lambda, \kappa_1) = 0$. This point can be used as the starting point for continuation applied to the combined system given by Eq. (3.4). The family of solutions to this system is the manifold of non-hyperbolic equilibria.

The above procedure illustrates two important situations. The first is that sophisticated continuation problems might involve the monitoring of values of certain functions; essentially waiting for a particular kind of event (in this case a zero crossing of $\psi(x, \lambda, \kappa_1)$). The second is that we often desire to constrain the values that a variable is *allowed* to take. This can be seen by the fact that in the first step, we set $\kappa = \kappa_1$, while in the second step, κ was allowed to vary freely.

We shall see in the next section that support for these typical tasks is a key benefit offered by COCO to users. It offers the ability to construct a continuation problem. It affords the flexibility of applying constraints at runtime. Using an interpreted language, it also allows for a high degree of interactivity with the problem solving process.

3.2 Problem Formulation

In this section we describe the process of formulating a COCO-compatible problem as described in [29]. Consider a function

$$F(x, \lambda, \kappa, \mu_1, \mu_2, \mu_3) = \begin{pmatrix} \phi(x, \lambda, \kappa) \\ \psi(x, \lambda, \kappa) \\ \lambda \\ \kappa \end{pmatrix} - \begin{pmatrix} 0 \\ \mu_1 \\ \mu_2 \\ \mu_3 \end{pmatrix} \tag{3.7}$$

If we determine the roots of $F(x, \lambda, \kappa, \mu_1, \mu_2, \mu_3)$ while restricting the value of $\mu_3 = \kappa_1$, the problem reduces to solving Eq. (3.6). Instead, if we restrict the value of $\mu_1 = 0$, solutions are elements of the set of non-hyperbolic equilibria. Therefore, our two stage problem can now be described succinctly as first solving Eq. (3.7) with $\mu_3 = \kappa_1$, then leaving the value of μ_3 unrestricted while setting $\mu_1 = 0$. Our changes at the level of restricting and releasing variables is transparent to the core functions; all they really see are parameters which are either restricted to certain domains or free.

We have now described a way to find the desired solution manifold of the system defined by Eq. (3.1). In the next section, we give an overview of the terminology used to describe such a process. The notation is from [9].

3.2.1 Continuation Problems

We shall make use of an important feature of COCO: the ability to *extend* a continuation problem with additional parameters and then *restrict* the problem by constraining the values of the added parameters. More formally, let

$$\Phi(u) = 0, \quad \Phi : \mathbb{R}^n \to \mathbb{R}^m, \quad n \geq m \geq 0 \tag{3.8}$$

be a continuously differentiable function. The equation $\Phi(u) = 0$ is known as a *zero problem* in the vector u of *continuation variables*. The m components of Φ are known as *zero functions*. Let $u = u^*$ be a regular point of the zero problem. As observed previously, the *Implicit Function Theorem* (IFT) guarantees the existence of a locally unique solution manifold in the neighborhood of u^*. The non-negative integer $n - m$ is termed as the *dimensional deficit* of the zero problem.

Now, let $\Psi : \mathbb{R}^n \to \mathbb{R}^r$ be another system of continuously differentiable functions, termed *monitor functions*, and let $\mu^* = \Psi(u^*)$. We construct a new system $F(u, \mu)$, where

$$F : (u, \mu) \to \begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \end{pmatrix} \tag{3.9}$$

and $\mu \in \mathbb{R}^r$ is a vector of *continuation parameters*. By construction,

$$F(u^*, \mu^*) = 0 \tag{3.10}$$

The corresponding solution manifold passing through the point (u^*, μ^*) will also be of dimension $n - m$.

We have just described how to extend the continuation problem; now we discuss how to restrict it. Let $\mathbb{I} \subseteq \{1, \ldots, l\}$ be an index set, $\mathbb{J} = \{1, \ldots, l\} \setminus \mathbb{I}$ and $\mu_{\mathbb{I}} = \{\mu_i | i \in \mathbb{I}\}$. The restriction is now obtained by setting $\mu_{\mathbb{I}} = \mu^*_{\mathbb{I}}$. The continuation problem with the imposed restrictions is known as a *restricted continuation problem*. Since the parameters $\mu_{\mathbb{I}}$ are *actively* imposing a constraint on the system, the corresponding equations are *active constraints*, and the parameters are said to be *inactive*, as they cannot change during continuation. The parameters $\mu_{\mathbb{J}}$ are called *active continuation parameters* since these variables are not restricted and hence can be used to monitor the values of certain functions of the system (the original use of the term *monitor* functions).

Summarizing, the problem construction process can be described as:

1. *Constructing* a list of functions $\Phi(u)$ such that we wish to solve for $\Phi(u) = 0$ in terms of the continuation variables u; this is the zero problem.

2. *Extending* the continuation problem by adding monitor functions $\Psi(u)$ to get an extended continuation problem $F(u, \mu)$, where μ are the continuation parameters.

3. Restrict the continuation problem by *constraining* the values of certain continuation parameters.

3.3 Constructing and Solving Problems Using COCO

In this section, we shall describe the basic syntax for constructing continuation problems using COCO. As mentioned earlier, COCO is written in MATLAB and hence all the data structures and syntax are that of MATLAB unless stated otherwise.

COCO encodes the continuation problem in a continuation problem structure, which is stored in a MATLAB `struct`. The syntax used to initialize this data structure is:

Listing 3.1: Initializing the encoding of an empty continuation problem

```
prob = coco_prob();
```

Functions in COCO must be described in a COCO-compatible format:

Listing 3.2: Function definition format for coco

```
function [data y] = fname(prob, data, u)
```

where `y` is the output of the functions acting on the vector of input arguments `u`. As an illustrative example, let us try to encode $\phi(x, \lambda, \kappa)$, defined in Eq. (3.1) in a COCO-compatible format:

Listing 3.3: coco-compatible encoding for Φ

```
function [data y] = phi(prob, data, u)
  y = u(3) - u(1)^2+u(1)^4+u(1)*u(2);
end
```

Here, we can see that `data` is both an input and output argument and changes made to this variable therefore survive execution. In more advanced applications, this structure will be used for encoding sophisticated operations; but in this simple function it is not used. Similarly, $\psi(x, \lambda, \kappa)$ defined in Eq. (3.3) is encoded in a COCO-compatible format as:

Listing 3.4: coco-compatible encoding for Ψ

```
function [data y] = psi(prob, data, u)
  y = -2*u(1) + 4*u(1)^3 + u(2);
end
```

Zero functions are added to the continuation problem using the following function call:

Listing 3.5: Adding zero functions

```
coco_add_func(prob, fid, fhan, data, 'zero', 'uidx', uidx, ...
  'u0', u0);
```

The inputs to this call are explained below:

- `prob` is a MATLAB struct that encodes the continuation problem;
- `fid` is a *function identifier*, a string label for the added function;
- `fhan` is a *function handle* to a COCO-compatible function;
- `data` is the *function data structure*, a MATLAB struct that contains the initial value of the `data` input argument to a COCO-compatible function. This can be an empty array if the function does not require such a structure;
- `'zero'` is a string label that indicates that the added function is a zero function;
- `uidx` is an index array of the subset of continuation variables defined thus far which are used in the function;
- `u0` is an array containing the initial solution guess for the newly added continuation variables.

Monitor functions can be added using a similar syntax:

Listing 3.6: Adding monitor functions

```
coco_add_func(prob, fid, fhan, data, 'inactive', pnames,...
  'uidx', uidx, 'u0', u0);
```

Here, `pnames` is cell array of string labels for the continuation parameters being added by the monitor function. These labels can subsequently be used when referring to the continuation parameters corresponding to this monitor

function. The continuation parameters added by the above function call is an *inactive* parameter, as specified by the fifth argument. Changing the string to `'active'` makes the parameter an *active* parameter.

We now construct a COCO formulation to implement the extended continuation problem corresponding to Eq. (3.7). We are given a point $P = (x_1, \lambda_1, \kappa_1)$ that lies on the manifold of equilibria, $\phi(x_1, \lambda_1, \kappa_1) = 0$.

We start by initializing the `prob` structure:

```
prob = coco_prob();
```

This function call initializes an empty continuation problem. We can now add the zero function $\phi(x, \lambda, \kappa) = 0$ (from Eq. (3.7))

```
coco_add_func(prob, 'phi', @phi, [], 'zero', ...
  'u0', [0.5 -1.625 1]);
```

Here, we use the label `'phi'` for the added function with function handle `@phi`. The `data` input argument is not used by the COCO encoding of $\phi(x, \lambda, \kappa)$ (as can be seen in listing 3.3), and hence can be empty. We do not specify `uidx` since the zero function added by this call does not use any of the previously added continuation variables (because there are none). However, the length of the initial value vector `u0` is used to create new continuation variables. In this case, the initial solution guess is $0.5, -1.625, 1$. Thus, three new continuation variables are associated with `prob`.

The next step is adding the monitor function $\psi(x, \lambda, \kappa)$ to the continuation problem:

```
coco_add_func(prob, 'psi', @psi, [], 'active', 'mu1', ...
  'uidx', [1:3]);
```

We have added the COCO-compatible function `psi` with a string label of `'psi'` and function handle `@psi`, and specified the continuation parameter as having the string label `'mu1'`. The `'active'` input string indicates that the continuation parameter μ_1 should be added to the set $\mathbb{J}$, i.e., that its value will not be restricted. The last two arguments specify that this monitor function uses the continuation variables that already exist. Here, `[1:3]` indicates that the first three continuation variables are chosen as the input arguments to the COCO-compatible function `psi`. Again, we see that the `data` argument remains empty; this is because the function `psi`, like the function `phi`, does not use the contents of `data` to compute the output value.

We now add the continuation parameters μ_2 and μ_3. In the first stage, we want to hold the value of $\mu_3 = \kappa_1$, while μ_2 can remain unrestricted. We use an encoding similar to the preceding one:

```
prob = coco_add_func(prob, 'iden1', @iden, [], 'active',...
  'mu2', 'uidx', [2]);
```

Here, `@iden` is a function handle to a COCO-compatible identity function:

Listing 3.7: coco-compatible encoding of the identity function

```
function [data y] = iden(prob, data, u)
  y = u;
end
```

For the first stage, we require the value of μ_3 to be fixed. Hence, we declare it as an inactive parameter (which implies an active constraint)

```
prob = coco_add_func(prob, 'iden2', @iden, [], 'inactive',...
  'mu3', 'uidx', [3]);
```

We have now constructed a continuation problem data structure `prob` that encodes the first stage of solution to the system described in section 3.2. Intuitively, we observe that the system $F(u, \mu) = 0$ can be equivalently described as system of two equations in three variables:

$$\psi(x, \lambda, \kappa) = 0 \tag{3.11}$$

$$\kappa = 0 \tag{3.12}$$

This is because only μ_3 was declared as an active constraint. The other continuation parameters do not restrict the system, they only reflect the value of certain functions of the continuation variables. The dimensional deficit of the system is one, therefore the solution manifold is a 1D branch.

We locate an initial point on the solution manifold and continue along the 1D solution branch using the `coco` entry point function:

```
coco(prob, 'run1', [], 1)
```

Here, `run1` is a label for this particular execution of the `coco` function. The last argument specifies the desired dimensional deficit of the problem (which here equals the dimensional deficit of the restricted continuation problem).

COCO allows one to observe the values of the continuation parameters as

the continuation progresses. This can be done by including the corresponding string labels in the call to the entry point function:

```
coco(prob, 'run1', [], 1, {'mu1' 'mu2' 'mu3'})
```

Executing this code will print the values of μ_1, μ_2, μ_3 to screen during continuation.

We are also interested in observing the values of the continuation variables, which in this case correspond to (x, λ, κ). This can be done by adding more continuation parameters, which simply track the values of these continuation variables. One way to do this would be to add monitor functions using the COCO-compatible `iden` function to each of the variables:

```
prob = coco_add_func(prob, 'iden3', @iden, [], 'active', 'x', ...
  'uidx', [1]);
prob = coco_add_func(prob, 'iden4', @iden, [], 'active', ...
  'lambda', 'uidx', [2]);
prob = coco_add_func(prob, 'iden5', @iden, [], 'active', ...
  'kappa', 'uidx', [3]);
```

and then specifying these variables in the call to `coco`:

```
coco(prob, 'run1', [], 1, {'x' 'kappa' 'lambda'})
```

COCO provides a function that makes this process simpler. This is the `coco_add_pars` function and it can be used to replace the code above as follows:

```
prob = coco_add_pars(prob, '' , 1:3, {'x' 'lambda' 'kappa'});
```

The third argument indexes into the array of continuation variables associated with the continuation problem structure `prob`. The last argument denotes the string labels that we wish to use for the added continuation parameters. The call to `coco` will remain the same.

The complete description of the first stage of the problem can now be given as:

Listing 3.8: coco code for first stage of problem 3.7

```
prob = coco_prob();
prob = coco_add_func(prob, 'phi', @phi, [], 'zero', ...
  'u0', [0.5 -1.625 1]);
prob = coco_add_func(prob, 'psi', @psi, [], 'active', ...
  'mu1', 'uidx', [1:3]);
prob = coco_set_parival(prob, 'mu3', 1);
prob = coco_add_pars(prob, '' , 1:3, {'x' 'lambda' 'kappa'});
```

```
coco(prob, 'run1', [], 1, {'x' 'kappa' 'lambda' 'mu1'})
```

Running this code gives an output:

```
  STEP   DAMPING              NORMS              COMPUTATION TIMES
IT SIT     GAMMA      ||d||     ||f||     ||U||   F(x)   DF(x)   SOLVE
 0                           3.57e-01  2.40e+00    0.0    0.0     0.0
 1   1  1.80e-01  5.55e-01  2.93e-01  2.62e+00    0.0    0.0     0.0
 2   1  2.34e-01  4.27e-01  2.10e-01  2.89e+00    0.0    0.0     0.0
 3   1  3.28e-01  3.05e-01  1.32e-01  3.19e+00    0.0    0.0     0.0
 4   1  5.26e-01  1.90e-01  5.79e-02  3.54e+00    0.0    0.0     0.0
 5   1  1.00e+00  8.31e-02  2.79e-10  3.86e+00    0.0    0.0     0.0
 6   1  1.00e+00  3.04e-10  0.00e+00  3.86e+00    0.0    0.0     0.0

STEP          ||U||  LABEL  TYPE       x        lambda        kappa          mu1
  0     3.8649e+00      1  EP    5.0000e-01  -1.6250e+00   1.0000e+00  -2.1250e+00
 10     7.7798e+00      2        2.5362e-01  -3.7056e+00   1.0000e+00  -4.1476e+00
 20     1.2712e+01      3        1.5643e-01  -6.2399e+00   1.0000e+00  -6.5374e+00
 30     1.7686e+01      4        1.1274e-01  -8.7589e+00   1.0000e+00  -8.9786e+00
 40     2.2672e+01      5        8.8046e-02  -1.1270e+01   1.0000e+00  -1.1444e+01
 50     2.7664e+01      6        7.2204e-02  -1.3778e+01   1.0000e+00  -1.3921e+01
 60     3.2658e+01      7        6.1184e-02  -1.6283e+01   1.0000e+00  -1.6405e+01
 70     3.7654e+01      8        5.3078e-02  -1.8787e+01   1.0000e+00  -1.8893e+01
 80     4.2651e+01      9        4.6867e-02  -2.1290e+01   1.0000e+00  -2.1384e+01
 90     4.7648e+01     10        4.1956e-02  -2.3793e+01   1.0000e+00  -2.3876e+01
100     5.2646e+01     11  EP    3.7976e-02  -2.6295e+01   1.0000e+00  -2.6370e+01

STEP          ||U||  LABEL  TYPE       x        lambda        kappa          mu1
  0     3.8649e+00     12  EP    5.0000e-01  -1.6250e+00   1.0000e+00  -2.1250e+00
 10     2.5132e+00     13        8.0063e-01  -9.6160e-01   1.0000e+00  -5.1004e-01
 20     4.5903e+00     14        1.2065e+00  -1.3787e+00   1.0000e+00   3.2338e+00
 30     9.2951e+00     15        1.4929e+00  -2.5041e+00   1.0000e+00   7.8186e+00
 40     1.4203e+01     16        1.6945e+00  -3.7608e+00   1.0000e+00   1.2311e+01
 50     1.9155e+01     17        1.8552e+00  -5.0691e+00   1.0000e+00   1.6762e+01
 60     2.4125e+01     18        1.9912e+00  -6.4054e+00   1.0000e+00   2.1190e+01
 70     2.9103e+01     19        2.1101e+00  -7.7593e+00   1.0000e+00   2.5602e+01
 80     3.4087e+01     20        2.2166e+00  -9.1253e+00   1.0000e+00   3.0005e+01
 90     3.9074e+01     21        2.3134e+00  -1.0500e+01   1.0000e+00   3.4399e+01
100     4.4063e+01     22  EP    2.4026e+00  -1.1882e+01   1.0000e+00   3.8787e+01
```

The points listed under the columns `x`, `lambda` and `kappa` give values of (x, λ, κ) for which $\phi(x, \lambda, \kappa) = 0$. The values of the continuation parameter `mu1` are of interest to us; we see that it crosses 0 between the output lines labeled 13 and 14. Thus, we can use either point as an initial guess $(x_2, \lambda_2, \kappa_2)$ for the second stage; as it is close to a point on the desired branch. (We could also rely on the COCO event-handling functionality, see part four of [] but will not do so here.)

The second stage of the problem is very similar to the first one except for two major differences:

1. the initial point is now $(x_2, \lambda_2, \kappa_2) = (0.80063, -0.96160, 1)$
2. μ_3 is left unrestricted but μ_1 is held at 0; this is done by declaring μ_1 as an inactive parameter and μ_3 as an active parameter

The dimensional deficit remains the same, as we *release* one continuation parameter while we *restrict* another one. Equivalently, we are now solving

the system:

$$\phi(x, \lambda, \kappa) = 0 \tag{3.13}$$

$$\psi(x, \lambda, \kappa) = 0 \tag{3.14}$$

where once again we have three variables but only two equations giving a 1D solution manifold.

The default value of μ_1 will be set by COCO to the value of $\psi(x_2, \lambda_2, \kappa_2)$. However, since we want the initial value of $\mu_1 = 0$, we use the COCO utility `coco_set_parival` to change the initial value of μ_1:

```
prob = coco_set_parival(prob, 'mu1', 0);
```

A listing of the solution is produced here for the sake of completeness; in the code we assign $(x_2, \lambda_2, \kappa_2) = (0.80063, -0.96160, 1)$, where we selected a point from the output of stage 1.

Listing 3.9: coco code for second stage of problem 3.7

```
prob = coco_prob();
prob = coco_add_func(prob, 'phi', @phi,...
  [], 'zero', 'u0', [8.0063e-01  -9.6160e-01   1.0000e+00]);
prob = coco_add_func(prob, 'psi', @psi,...
  [], 'inactive', 'mu1', 'uidx', [1:3]);
prob = coco_add_func(prob, 'iden1', @iden,...
  [], 'active', 'mu2', 'uidx', [2]);
prob = coco_add_func(prob, 'iden2', @iden,...
  [], 'active', 'mu3', 'uidx', [3]);
prob = coco_set_parival(prob, 'mu1', 0);
prob = coco_add_pars(prob, '' , 1:3, ...
  {'x' 'lambda' 'kappa'});
coco(prob, 'run1', [], 1, {'x' 'lambda' 'kappa' 'mu1'})
```

Running this code gives an output:

```
  STEP   DAMPING               NORMS              COMPUTATION TIMES
IT SIT     GAMMA      ||d||      ||f||      ||U||   F(x)   DF(x)   SOLVE
 0                             1.39e-01  2.66e+00    0.0     0.0     0.0
 1   1  1.00e+00  3.67e-02  2.06e-02  2.70e+00    0.0     0.0     0.0
 2   1  1.00e+00  9.01e-03  2.01e-04  2.70e+00    0.0     0.0     0.0
 3   1  1.00e+00  1.19e-04  1.54e-08  2.70e+00    0.0     0.0     0.0
 4   1  1.00e+00  1.02e-08  7.01e-17  2.70e+00    0.0     0.0     0.0

STEP          ||U||  LABEL  TYPE         x        lambda        kappa          mu1
  0     2.6960e+00      1  EP    8.7728e-01  -9.4611e-01   1.0073e+00   1.1102e-16
 10     1.1219e+00      2        5.0257e-01   4.9739e-01  -6.1193e-02   0.0000e+00
 20     1.1127e+00      3        4.0524e-01   5.4429e-01  -8.3315e-02   0.0000e+00
 30     9.9764e-01      4        3.1956e-01   5.0859e-01  -7.0835e-02   5.4323e-13
 40     4.7777e-01      5       -1.3134e-01  -2.5361e-01  -1.6357e-02   0.0000e+00
 50     1.1123e+00      6       -4.0483e-01  -5.4427e-01  -8.3310e-02   0.0000e+00
 60     1.1325e+00      7       -4.7508e-01  -5.2126e-01  -7.2879e-02  -2.4347e-13
 70     1.0046e+00      8       -6.5771e-01  -1.7737e-01   1.2880e-01   1.1102e-16
 80     4.9412e+00      9       -9.9334e-01   1.9339e+00   1.9341e+00  -2.9925e-12
 90     9.8657e+00     10       -1.1578e+00   3.8921e+00   4.0499e+00  -2.4869e-14
```

```
100     1.4826e+01     11  EP   -1.2740e+00   5.7228e+00   6.2796e+00   0.0000e+00

STEP          ||U||  LABEL  TYPE         x        lambda        kappa          mu1
  0     2.6960e+00     12  EP    8.7728e-01  -9.4611e-01   1.0073e+00   1.1102e-16
 10     6.7035e+00     13        1.0611e+00  -2.6570e+00   2.6775e+00   3.8325e-13
 20     1.1649e+01     14        1.2033e+00  -4.5629e+00   4.8419e+00   7.1054e-15
 30     1.6612e+01     15        1.3092e+00  -6.3583e+00   7.1005e+00   8.8818e-16
 40     2.1580e+01     16        1.3952e+00  -8.0734e+00   9.4213e+00   3.5527e-15
 50     2.6550e+01     17        1.4683e+00  -9.7254e+00   1.1788e+01   1.7764e-15
 60     3.1521e+01     18        1.5323e+00  -1.1326e+01   1.4190e+01  -1.7764e-15
 70     3.6491e+01     19        1.5894e+00  -1.2883e+01   1.6620e+01   0.0000e+00
 80     4.1462e+01     20        1.6413e+00  -1.4402e+01   1.9075e+01   1.7764e-15
 90     4.6433e+01     21        1.6888e+00  -1.5888e+01   2.1549e+01   1.7764e-15
100     5.1404e+01     22  EP    1.7327e+00  -1.7344e+01   2.4041e+01   0.0000e+00
```

The output shown before the actual continuation is that of the Newton solver as it determines a solution point near the initial guess. Once it has determined such a solution point, the continuation algorithm is started at that point. Here, we observe that the value of `mu1` always remains 0. The set of points (x, λ, κ) for which this holds true are precisely the solution points to the system given by Eq. (3.4).

We have provided an introduction to the construction and manipulation of continuation problems in COCO. There are many other COCO-compatible utilities designed to make the process of problem construction and solving automatic and programmable; these are described in [9].

3.4 Continuation in COCO

The COCO toolbox supports two very important features. Firstly, it allows us to *construct* an extended continuation problem and then to *constrain* the problem at runtime. This was described in the previous section. The other important feature is the ability to *solve* the constructed continuation problem and determine a manifold of its solutions.

We described the theory of continuation in section 2.1. Here, we shall describe the implementation in COCO of the theoretical ideas discussed in that section. The terminology used to describe these ideas comes from [9].

3.4.1 Theory

In this section, we briefly describe the theoretical concepts that are extensively used as part of COCO's manifold covering algorithms. We shall see that many of these concepts have similarly named structures representing them

in COCO.

We consider the case of a continuously differentiable continuation problem $\Phi(u) = 0$. This is the same as finding the roots of the system of functions $\Phi : \mathbb{R}^n \to \mathbb{R}^m$ for $n \geq m \geq 0$. If u^* is a regular solution point, the IFT guarantees the existence of a locally unique solution manifold of dimension $n - m$.

If we find a sufficient number of tangent vectors to the solution manifold at the point u^*, we can use them to compute a basis for the space of tangent vectors, or the *tangent space*, at the point. The tangent space can then be used to provide a direction in which to search for a new solution point. This direction may be used to compute an initial guess (which is known as a *predictor*). We hope to get a guess that is close to the solution manifold. This guess is then used to search for a point on the solution manifold; the algorithm that does so is known as a *corrector* algorithm. Once such a point is obtained, we repeat the same process until some termination condition is reached.

We can represent a curve on the solution manifold through the point u^* by the function $\gamma : [-\eta, \eta] \to \mathbb{R}^n$, for some small η such that $\gamma(0) = u^*$. All such *curve segments* must satisfy the property:

$$\Phi(\gamma(t)) \equiv 0, t \in [-\eta, \eta] \tag{3.15}$$

For each curve segment, $\gamma'(0)$ represents the *tangent vector* at the point u^*. The space of all such tangent vectors is called the *tangent space* $\mathcal{T}_{u^*}$ of the solution manifold at the point u^*.

It can be shown that the nullspace of the Jacobian $\partial_u \Phi(u^*)$ corresponds to the tangent space $\mathcal{T}_{u^*}$ [9]. The problem of determination of the tangent space then reduces to the problem of finding the null vectors of the Jacobian. Let $V \in \mathbb{R}^{n \times (n-m)}$ be an arbitrary matrix whose columns consist of orthonormal vectors

$$V^T \cdot V = I_{n-m} \tag{3.16}$$

and for which the square matrix

$$\begin{pmatrix} \partial_u \Phi(u^*) \\ V^T \end{pmatrix} \tag{3.17}$$

is invertible. Then the tangent space $\mathcal{T}_{u^*}$ at the point u^* is spanned by the columns of the matrix

$$\begin{pmatrix} \partial_u \Phi(u^*) \\ V^T \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 \\ I_{n-m} \end{pmatrix} \tag{3.18}$$

Let $V^\perp \in \mathbb{R}^{n \times m}$ be a matrix whose columns are linearly independent and which satisfies the equation:

$$V^T \cdot V^\perp = 0 \tag{3.19}$$

It can be shown that there exists a smooth function $\lambda : \mathbb{R}^n \to \mathbb{R}^m$ such that

$$\lambda(u^*) = 0 \tag{3.20}$$

and such that $\tilde{u} + V^\perp \cdot \lambda(\tilde{u})$ lies on the solution manifold for every point $\tilde{u}$ on some neighborhood of u^*. We can now formulate a closed continuation problem as:

$$\begin{pmatrix} \Phi(u) \\ V^T \cdot (u - \tilde{u}) \end{pmatrix} = 0 \tag{3.21}$$

Note that the point $u := \tilde{u} + V^\perp \cdot \lambda(\tilde{u})$ is the locally unique solution to the above problem. The equation

$$V^T \cdot (u - \tilde{u}) = 0 \tag{3.22}$$

is known as the $n - m$ dimensional *projection condition*. It can be seen that the projection condition is invariant under a translation $\tilde{u} \mapsto \tilde{u} + V^\perp \cdot \mu$, for an arbitrary $\mu \in \mathbb{R}^m$. Hence, the solution manifold through the point u^* can be described locally by the function:

$$\rho \mapsto \tilde{u} + V \cdot \rho + V^\perp \cdot \lambda(\tilde{u} + V \cdot \rho) \tag{3.23}$$

for some *fixed* $\tilde{u}$ and $\rho \in \mathbb{R}^{n-m}$ on some neighborhood of u^*. Here, we are using the fact that

$$\operatorname{span}(V) \oplus \operatorname{span}(V^\perp) = \mathbb{R}^n \tag{3.24}$$

to express a point $u \in \mathbb{R}^n$ in terms of V and $V^\perp$.

It can be shown that, for every $\tilde{u}$ in a small neighborhood of u^*, there

exists a unique ρ^* such that:

$$u^* = \tilde{u} + V \cdot \rho^* + V^\perp \cdot \lambda(\tilde{u} + V \cdot \rho^*) \tag{3.25}$$

Multiplying both sides by V^T, we get

$$V^T \cdot (u^* - \tilde{u}) = \rho^* \tag{3.26}$$

Suppose that we are only interested in finding the solution branches characterized by a fixed direction, associated with a unit vector $s \in \mathbb{R}^n$. We set $\rho^* = hs$, where $h = \|\rho^*\|$. If we now make the substitution

$$\tilde{u} := \tilde{u} + V \cdot hs + V^\perp \cdot \lambda(\tilde{u} + V \cdot hs) \tag{3.27}$$

in Eq. (3.26), we get

$$V^T \cdot (u - \tilde{u}) - h \cdot s = 0 \tag{3.28}$$

We note that this formulation of the projection condition gives points u in a branch of the solution manifold and not the complete manifold itself. This is because s is fixed; only its magnitude h is permitted to change.

If we have a solution point u^* and desire to search for more solution points in a tangent direction t, we must assign a value to s that satisfies the following conditions:

$$s = \frac{V^T \cdot t}{\|V^T \cdot t\|} \tag{3.29}$$

$$s \quad \| \quad V^T \cdot (u^* - \tilde{u}) \tag{3.30}$$

where the latter condition only applied if $\tilde{u} \neq u^*$. The curve segment defined in Eq. (3.28) is then said to be a *continuation* of the point u^* in a tangent direction $t \in \mathcal{T}_{u^*}$.

We can now define a *predictor* as an algorithm that produces an intial solution guess using $\Phi(u)$, u^*, $\tilde{u}$, V, h and s. As an example, a linear predictor is given by $u = \tilde{u} + hV \cdot s$. Given such a predictor, a *corrector* algorithm is used to obtain a point on the solution manifold. Typically, the corrector algorithm is a nonlinear iterative equation solver, like the Newton method.

We have thus defined all the theoretical concepts that will be required in our discussion on the covering of manifolds using atlas algorithms. In the

next section, we give an overview of what the implementation of these ideas looks like in COCO.

3.4.2 A Finite State Machine For Atlas Algorithms

We have seen that PSALC methods describe the manifold with the help of compatible simplices. By default, COCO uses PSALC based-methods, where these simplices are represented by *charts*. A chart $\{u, T, \Sigma, R\}$ contains information about the following properties:

1. the base point u;
2. a matrix T which consists of orthonormal basis vectors of the tangent space of u;
3. a set of coordinate vectors Σ in the basis of T that indicate possible directions of further continuation;
4. a scalar R which gives the extent of the domain over which this representation is assumed to be valid.

A family of such charts is called an *atlas*. An atlas, whose base points are sufficiently dense over a manifold, is said to *cover* it. An *atlas algorithm* is an algorithm that determines such a covering. It consists of two stages: *expansion* and *consolidation*. In the expansion stage, a chart is used as the base to create a sequence of charts along a direction given by some element of Σ. We use a *projection condition* to locate such points on the solution manifold. In the consolidation stage, this sequence of charts is merged into the atlas. The projection condition and the associated sequence of charts are said to constitute a *curve segment*.

Atlas algorithms can be described with the help of Finite State Machines (FSM). Using a FSM allows us to partition the iterative solution process into different, discrete states. *Transition functions* then specify the conditions under which the state must change. These transition functions operate on the existing atlas and curve segment to determine the next state. The details of this process will be described in the next section.

The COCO framework allows the user to specify different atlas algorithms to obtain the covering. Each such atlas algorithm may be represented by an FSM associated with it. In this section, we describe the simplest such FSM.

The atlas algorithm starts at the `init` state and then enters a loop between the `flush`, `predict`, `correct` and `add` states. The computation exits the loop when a transition from the flush state to the `stop` state is triggered.

The computation starts in the `init` state. In this state, the initial chart is created and a partial curve segment is initialized. A tangent matrix V is computed and the intial guess is assigned to the base point $\tilde{u}$. The initial solution guess is then used to find a point u that satisfies projection condition:

$$V^T \cdot (u - \tilde{u}) = 0 \tag{3.31}$$

The point u will thus lie on the solution manifold. The matrix V is then used to compute the tangent matrix T, and a set of coordinate vectors is assigned to Σ. Thus, a chart has been computed; this chart is used as the initial chart of the curve segment.

The `predict` state requires a partial atlas as input, which is constructed in the preceding `flush` state. In this state, a chart from the atlas is chosen as the base chart; again, we initialize a partial curve segment. This is used to compute a new base point $\tilde{u}$, a *tangent matrix* V, a *direction vector* s and a *step size* h corresponding to the projection condition:

$$V^T \cdot (u - \tilde{u}) - hs = 0 \tag{3.32}$$

Finally, we initialize a corrector. As discussed in section 2.1, this is usually a nonlinear iterative solver that uses the predictor as an initial guess to locate a point on the solution manifold.

The `correct` state applies the corrector algorithm to locate a new point u^* on the solution manifold which also satisfies the projection condition.

The `add` state is entered by the algorithm with a partial curve segment constructed in the `predict` state, and a new solution point u^* along this curve segment constructed in the correct state. This state constructs a new chart using u^* and appends it to the partial curve segment.

Finally, the `flush` function merges any partial curve segments into the atlas. It does so by modifying the content of the set of coordinate vectors Σ for the charts in the atlas.

This concludes our discussion of the FSM used within COCO for implementing atlas algorithms. The details of the computation in each state are

deliberately not specified in our description; for every specific atlas algorithm, there exists a concrete implementation of each state.

The representation of an atlas algorithm as an FSM thus provide us an abstract interface to the problem of writing atlas algorithms for COCO. In the next section, we shall specify the COCO-compatible syntax of this interface.

3.5 Implementation of Atlas Algorithms

The atlas algorithms used in COCO are designed using an object-oriented paradigm. An object oriented design is one where the units of computation are represented by *objects*, which can be thought of as structures that contain both data and functions (called *methods*) which operate on the data. Objects are designed by specifying their data and methods by an abstract structure known as a *class*. Every object is an *instance* of its class. A class may *inherit* from another class, in which case it has all the structure of its parent class. However, it is free to override the method definitions (unless specifically forbidden by language features) and declare its own methods. In this relationship of inheritance, the class that inherits is known as a *subclass* of the parent class.

Data can be stored by an object; it is known as a *property* of the object. Both properties and methods of an object can be *private* in which case they are visible only to instances of the same object; or *public*, in which case they are visible to all objects in the system. By visible, we mean that the corresponding property or method can be accessed by other objects.

Fundamentally, object-oriented programming allows for a better organization of the computational process and permits the creation of abstract *interfaces*, which are the description of the actions that can be performed by an object. Specifying an interface allows the creation of easily extendable programs; a user needs only use the correct interface for his program and integrate this easily within a larger computational system.

We see a requirement of precisely this nature for COCO. In this chapter, we are concerned with using different kinds of atlas algorithms with minimal rewrite or change of COCO code. Specifying the important structures as abstract classes with interfaces allows the creation of a blueprint which all atlas algorithms must follow.

We describe two classes that are used by COCO to encode arbitrary atlas algorithms. These are the `AtlasBase` class and the `CurveSegment` class. Customized atlas algorithms are constructed by subclassing from the `AtlasBase` class, with the possibility of both overriding existing class methods and adding additional such methods.

The charts introduced in the previous section are represented by the `chart` structure. This structure encodes a few reserved fields. However, any number of additional fields can be added by the atlas algorithm.

Any subclass to the `AtlasBase` class must have at least the following structure to be compatible with COCO:

Listing 3.10: Minimal structure of a coco-compatible `AtlasBase` class

```
classdef algorithm_kd < AtlasBase

% A blueprint for all atlas algorithms used in COCO

properties (Access=private)
% global properties used by the atlas algorithm
% are defined here
end


methods (Access=private)
  % class constructor for the subclass algorithm
  function atlas = algorithm_kd(prob, cont, dim)
    % the constructor for the base class is
    % called first
    atlas = atlas@AtlasBase(prob);
    % ... additional commands .... %
  end

end

methods (Static) % construction method
  function [prob cont atlas] = create(prob, cont, dim)
    % call the class constructor for the subclass algorithm
    atlas = algorithm_kd(prob, cont, dim);
    % add the projection condition to the prob structure
    prob = CurveSegment.add_prcond(prob, dim);
    % ... additional commands .... %
  end

methods (Access=public)
% define class methods corresponding to the
% different states in the atlas algorithm
  function [prob atlas cseg correct] = ...
      init_prcond(atlas, prob, chart)
    [prob cseg] = ...
      CurveSegment.create_initial(prob, chart);
```

```
    correct = cseg.correct;
    % ... additional commands .... %
  end

  function [prob atlas cseg flush] = ...
      init_atlas(atlas, prob, chart)
    % ... additional commands .... %
  end

  function [prob atlas cseg correct] = ...
      predict(atlas, prob, chart)
    % ... additional commands .... %
  end

  function [prob atlas cseg flush] = ...
      add_chart(atlas, prob, chart)
    % ... additional commands .... %
  end

end

end
```

The `algorithm_kd` class constructor is invoked indirectly by a call to the `create` method (A *constructor* of a class is a method that instantiates an object of the class). The input argument `dim` contains the intended dimensional deficit of the restricted continuation problem. The `prob` input argument is the continuation problem structure. The `cont` input argument is a MATLAB `struct` that holds any other setting that may be given to the atlas algorithm. Its fields may be set using the `coco_set` function as follows:

```
prob = coco_set(prob, 'cont', 'property', value)
```

For example, we often want to change the maximum number of points that may be computed by the atlas algorithm before termination; this can be done by specifying the `PtMX` field of the `cont` structure as follows:

```
prob = coco_set(prob, 'cont', 'PtMX', 100)
```

If the atlas algorithm reads this value and sets this property on its own `cont` property, the computation will terminate after 100 solution points are computed. The `AtlasBase` class also provides default implementation of the following methods, which may be replaced by the subclass with its own implementation:

```
[prob atlas cseg correct] = init_prcond (atlas, prob, chart)
[prob atlas cseg flush  ] = init_atlas  (atlas, prob, cseg)
[prob atlas cseg flush  ] = add_chart   (atlas, prob, cseg)
```

```
[prob atlas cseg correct] = predict     (atlas, prob, cseg)
```

The methods `add_chart`, `predict`, `flush`, `init_atlas` and `init_prcond` correspond to states in the FSM. In particular, `add_chart` corresponds to the `add` state, `predict` corresponds to the `predict` state; `init_atlas` and `init_prcond` refer to states introduced in a more sophisticated version of the `init` state (described in [9]). More states can be specified by adding similar methods as class methods of `algorithm_kd`. We shall see examples of atlas subclasses in the next chapter. Note that the every atlas algorithm that we talk about in this thesis uses COCO's default implementation of the `correct` state, which is a nonlinear iterative Newton solver.

The sequence of states traversed by the FSM can be altered in two different ways. The first one is by changing a boolean flag that is present in the output argument list of a class method. The next one is by altering the value of the `cseg.Status` property. For example, the boolean output argument `correct` of the `predict` method is used to determine the next state. If it is true, then it means that the atlas algorithm will enter the `correct` state.

The `CurveSegment` class provides a representation of the curve segment concept that we introduced earlier. One of its important functions is to record the projection condition. Each instance of the `CurveSegment` class includes:

- `ptlist`: 1D cell array of charts along the curve segment;
- `src_chart`: parent chart along the curve segment;
- `curr_chart`: working copy of current chart;
- `prcond`: MATLAB `struct`, representing the projection condition, that contains:
  - `x`: base point $\tilde{u}$;
  - `TS`: tangent matrix V;
  - `s`: direction vector s;
  - `h`: step size h.
- `Status`: integer flag used by the atlas algorithm to decide whether to merge a curve segment into the atlas;

- `correct`: boolean flag representing whether the corrector algorithm should be applied to the closed continuation problem.

The `cseg` input and output argument of the various class methods defined earlier represent instances of this class. We instantiate an instance of the `CurveSegment` class in the `init_prcond` class method of an `AtlasBase` subclass with the help of the following static method (a `static` method is a method which, although associated with a class, does not require an instance of the class as input):

```
[prob cseg] = CurveSegment.create_initial(prob, chart);
```

There are two important class methods associated with the `CurveSegment` class. The `CurveSegment.create_initial` class method instantiates an instance of the class with the following properties:

- `ptlist`: an empty array;
- `prcond.x`: `chart.x`;
- `prcond.TS`: a tangent matrix V;
- `prcond.s`: unit vector $w \in \mathbb{R}^{n-m}, w(1) = 1$;
- `prcond.h`: 0;
- `src_chart`: `chart`;
- `curr_chart`: a partial copy of `chart`, consisting of x, R and pt fields, along with any non reserved fields defined on the chart.

The `CurveSegment.create` instantiates a `CurveSegment` object. This is done in the `predict` class method of the `AtlasBase` class, with the following call:

```
[prob cseg] = CurveSegment.create(prob, chart, prcond, x);
```

This method differs from `CurveSegment.create_initial` in the following manner:

- the `chart` argument is assigned as a single element of the array stored in the `ptlist` property;
- `prcond` argument is assigned to the `prcond` property;

- the `x` argument is assigned to the `x` field of the `curr_chart` property.

In both methods, the `curr_chart` field provides an initial solution guess which is then refined by the corrector algorithm. If the corrector algorithm converges, this field is updated to the newly located solution point.

Thus, we now have a basic understanding of how the COCO toolbox works. We shall rely on the concepts introduced in this chapter to describe specific atlas algorithms in the next chapter.

CHAPTER 4

PARALLEL MANIFOLD COVERING

In this chapter, we shall go into details about COCO's implementation of atlas algorithms, as well as the changes that we made to include parallelism.

We begin in Section 4.1 with a brief overview of the terminology used to describe atlas algorithms. We have discussed this in detail in Section 3.5. Our discussion on atlas algorithms begins in earnest in Section 4.2.1, where we discuss 1D atlas algorithms; then in Section 4.2.2, we discuss the 2D versions. Finally, in Section 4.3, we discuss our implementation of parallel versions of atlas algorithms. We conclude with Section 4.4, where we enumerate the results of our experiments with the algorithms defined in the previous Sections.

4.1 Terminology

In this section, we define a correspondence between the theoretical concepts introduced in Section 3.4.1 and the structures used to represent them in COCO.

A *chart* is represented by a MATLAB `struct` with the following fields:

- a *base point* u;
- the *tangent matrix* $\mathcal{T}$, whose columns constitute an orthonormal basis of the tangent space of u;
- a set Σ of coordinate vectors (in the basis given by the columns of $\mathcal{T}$) of candidate unit vectors at u along the solution manifold;
- a scalar R that represents the distance between u and the next point to be computed.

A family of such charts is called an *atlas*, like before. A *cover* of the solution manifold is obtained by computing an atlas whose base points are sufficiently dense on this part of the manifold. An *atlas algorithm* generates an atlas using an initial point u and a tangent matrix V. In every step, a base chart is used to construct a sequence of charts along the direction $T \cdot \gamma(\sigma)$ for some $\sigma \in \Sigma$ and a function γ; this sequence of charts is known as a *curve segment*. The curve segment is then merged into the atlas.

4.2 Atlas Algorithms

In this section, we introduce the different atlas algorithms that are described in [9]. We shall attempt to summarize the description; in particular, we shall describe the theory that defines the atlas algorithms and make note of the major differences between them. A thorough and very complete description of these algorithms (including descriptions of individual states and state transition functions) can be found in [9].

4.2.1 1D Atlas Algorithms

We discuss the two kinds of atlas algorithms for covering 1D manifolds as described in [9].

Consider the continuously differentiable continuation problem

$$\Phi(u) = 0, \quad \Phi : \mathbb{R}^n \to \mathbb{R}^{n-1} \tag{4.1}$$

for $n \geq 2$. If u^* is a regular solution point, ($\Phi(u^*) = 0$) and the rank of the Jacobian $\partial_u \Phi(u^*)$ is $n-1$, then the Implicit Function Theorem (IFT) implies the existence of a locally unique solution manifold through the point u^*.

If $v \in \mathbb{R}^n$ is an arbitrary unit vector such that the square matrix

$$\begin{pmatrix} \partial_u \Phi(u^*) \\ v^T \end{pmatrix} \tag{4.2}$$

is invertible, then the 1D tangent space at u^* is spanned by the vector

$$t^* = \begin{pmatrix} \partial_u \Phi(u^*) \\ v^T \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4.3}$$

Given a base point u^*, direction vector s and step size h, the 1D projection condition is given by

$$v^T \cdot (u - u^*) - hs = 0 \tag{4.4}$$

4.2.1.1 Advancing Local Cover

The *Advancing Local Cover* method is a PSALC method where the predictor is given by $u^* + Rst^*$ for $s \in \{-1, 1\}$ and some scalar R. The tangent space at u^* is spanned by the unit vector t^*. Here, the tangent matrix for projection condition v equals t^* and corresponding base point u equals u^*. The tangent space at a point u along a curve segment is then spanned by the vector

$$t := \begin{pmatrix} \partial_u \Phi(u) \\ t^{*T} \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4.5}$$

The continuation is ensured to proceed in a fixed direction t by the imposition of the restriction $t^{*T} \cdot t = 1$. Since the tangent direction $t = st^*$, the basis set, $\Sigma = \{-1, 1\}$.

The algorithm is initialized with a base point u_0 and a suitable guess for the initial tangent matrix v. The determination of each successive solution point u^* is followed by the construction of an associated chart $\{u^*, t^*, s, R\}$ and continuation proceeds in the direction of st^*. This implementation tracks only the most recently located solution point; hence, the method is known as *advancing local cover*.

The algorithm terminates when:

- the number of computed points exceeds the maximum specified limit;
- the corrector algorithm fails to converge.

Since the algorithm tracks only the most recently located solution point, it is possible that it will trace closed solution manifolds repeatedly.

4.2.1.2 Adaptation And Accelerated Convergence

In the advancing local cover method described in the previous section, the step size h is kept fixed in each step of the continuation. The method gives us a set of points along the solution manifold; these points are said to represent the 1D solution manifold. We can think of this as a discretization of the solution manifold. In regions where the solution curve is flat, we need fewer points to represent it. Conversely, in regions with higher curvature, more points are required to construct a faithful representation of the manifold. Therefore, if we had a mechanism to *adapt* the step size with the local curvature, we could get a more efficient representation of the solution manifold. This is possible because we generate fewer points where the curve is flat, while spending time in constructing more points in regions of high curvature.

In [9], an implementation is presented which aims to do precisely what we described. We present this method below.

This method includes two important additions over the previous algorithm. First, the tangent vector v of the projection condition is obtained as:

$$v := \begin{pmatrix} \partial_u \Phi(u^* + \theta R s t^*) \\ t^* \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4.6}$$

where $\theta \in [0, 1]$. The step size h is now given by $Rv^T \cdot t^*$ and the new predictor is given by the linear predictor $u^* + hsv$.

The next change that we make is in adaptively changing the scalar radius R depending upon the success or failure of the corrector algorithm. If the corrector algorithm in the `correct` state fails to converge, the algorithm enters the `refine` state, where the predictor is recomputed with a smaller value of the scalar radius R. The scalar radius is multiplicatively reduced, until the correction algorithm converges, or it reaches a predefined minimum value. This can be seen as an attempt to search for points along the solution manifold that lie closer to the regular point along the curve segment defined by the projection condition.

Conversely, in the `add_chart` state, the scalar radius R is increased upon successful convergence, up to a predefined maximum value. The new tangent vector is now checked for alignment with the old tangent vector; if the deviation exceeds a value $\alpha_{\max}$, then this can lead to two outcomes:

1. if the radius R is greater than its minimum allowed value, then R is reduced and the continuation is repeated with the same chart;

2. if the radius R is smaller than its minimum allowed value, then the chart is accepted as a valid chart.

4.2.1.3 Expanding Boundary Algorithm

The Advancing Local Cover algorithm discussed in section 4.2.1.1 represents the atlas by a single base chart at the head of an advancing local cover along the solution manifold. In the expansion phase, a single chart is constructed in one direction; in the consolidation phase the new chart replaces the old chart.

However, for a 1D manifold, we observe that its boundary is made of *two* points, not one. The interior of the solution manifold lies in between these two points. Therefore, a better representation of the 1D manifold could be obtained by storing both these points.

In this section, we discuss such an implementation where an atlas is represented by the charts on its boundary which enclose an expanding 1D curve of solutions. The expansion phase now consists of continuation in an outward direction from the atlas interior; the newly generated charts are then consolidated into the atlas.

In section 4.2.1.2, we argued that the collection of points computed by the atlas algorithm could be viewed as a representation of the solution curve. We can make this representation more explicit by requiring that there be a line segment associated with each point lying along the tangent line to the solution curve at that point; the covering of the solution manifold now consists of a set of local line segments, whose projections onto the solution manifold overlap locally near their end points.

In the first algorithm described in this section, the step size used for continuation was not an adaptive one. Therefore in regions of high curvature, the step size used by the algorithm may be high, leading to *gaps* in the covering: regions of the solution manifold that are not covered by any chart, but lie in between them. The second algorithm gave some improvement in this area by varying the continuation step size to ensure good coverings in regions of high curvature.

In this section, we impose the following conditions (as listed in [9]) that two charts must satisfy in order to be considered as neighbors:

- the distance between the projection of the base point of one chart into the tangent space of the other should be less than R;
- each base point should lie inside a cone centered at the base point of the other chart. This cone is aligned along the tangent vector of the other chart and has an opening angle of $2\alpha_{\max}$, where $\alpha_{\max}$ is the angle that was introduced in section 4.2.1.2;
- the tangent spaces of the charts must not be out of alignment by more than $\alpha_{\max}$.

In short, the two points must lie in a conical region close to each other. We can mathematically represent these criteria as follows. Let u_1 and u_2 be the two base points of the charts being merged with tangent spaces spanned by unit vectors t_1 and t_2 respectively. Then, if $du = u_2 - u_1$, the two charts are said to be neighbors provided that

$$|t_1^T \cdot du| < R \tag{4.7a}$$

$$|t_2^T \cdot du| < R \tag{4.7b}$$

$$\|du - t_1(t_1^T \cdot du)\| < |t_1^T \cdot du| \tan \alpha_{\max} \tag{4.7c}$$

$$\|du - t_2(t_2^T \cdot du)\| < |t_2^T \cdot du| \tan \alpha_{\max} \tag{4.7d}$$

$$t_1^T \cdot t_2 > \cos \alpha_{\max} \tag{4.7e}$$

These conditions are used to determine whether a newly computed chart during continuation is a neighbor of the existing boundary charts. If it is not, then either the chart is recomputed, or the algorithm must terminate. Once a chart is determined to be a neighbor of a boundary chart, the new chart replaces the old boundary chart. Since $\Sigma = \{-1, 1\}$, we now have an *expanding boundary* of charts at the endpoints of the solution branch.

4.2.2 2D Atlas Algorithms

We now discuss algorithms for covering 2D manifolds that are given in [9]. There are multiple directions that a 2D solution manifold can be continued

along. Hence, one important issue is that of selecting directions of continuation, especially since we do not know what the solution manifold looks like. It should be noted that the boundary of the solution manifold, which in 1D could be represented by the two end-points of the solution curve, is now more complicated: it will be a connected 1D chart network of neighboring charts that enclose the region of the manifold covered by the atlas algorithm. We must also choose directions of atlas expansion such that we don't repeatedly cover the same regions of the manifold.

4.2.2.1 Point Cloud Method

We introduce a straightforward extension of the expanding boundary algorithm described in section 4.2.1.3 to work in higher dimensions (as given in [9]). In particular, instead of limiting $s \in \{-1, 1\}$, we construct Σ such that the corresponding direction vectors are uniformly oriented in the tangent space of the chart.

We face two important issues with this simple algorithm, both related to the poor representation of the atlas boundary. The first is the problem of early termination. If the manifold being covered is a closed manifold, then there is a possibility that the algorithm will trace out a closed curve and then terminate. For example, as shown in [9], for a spherical solution manifold and a certain choice of Σ, the algorithm traces out a great circle and then terminates even though the rest of the solution manifold remains undetermined.

The next problem is that of redundant covering. In the above algorithm, charts are dropped from the boundary array once all their predictors are exhausted; effectively, the algorithm forgets about the existence of such a chart. Once this happens, the same region may be covered again, since there is no other mechanism to detect overlap.

Therefore, in the absence of any mechanism to check existing covering, this algorithm might cover the same region repeatedly resulting in a cloud of base points, instead of an efficient and sparse cover using the minimal set of charts. In particular, we note that the availability of multiple directions of continuation makes redundant coverage the norm, rather than the exceptional case it was in 1D.

4.2.2.2 Chart Network Method

The *Chart Network Method*, as presented in [9], attempts to remedy the problem of redundant coverage. It does so by storing all the charts constructed by the atlas algorithm. The charts are stored in the order that they are constructed; this order can also be deduced from the unique ordinal number assigned to each chart. The boundary charts of the atlas are stored in a list of ordinal numbers that index into the array of charts; indicating which charts are known to form the boundary of the solution manifold.

The `chart` structure now contains two more properties of interest to us. First, the `nb` property is an array of integers which gives the ordinal numbers of the charts that are in the neighborhood of this chart. Thus, we have a representation of the manifold in terms of a list of charts that cover it, along with local neighborhood information stored with each individual chart. The second important property is the `bv` property. This is also an array of integers, and it indexes into Σ to give the subspace of tangent vectors, that can be used for atlas expansion. By construction, the boundary array of the atlas contains only those charts that have a nonempty `bv` array. Thus, whenever a new chart is merged into the atlas, all the charts indexed by the boundary array are examined to see if they have nonempty `bv` values after the merge, and the corresponding indices remain in this array only when this is the case.

When a new chart is added to the atlas, a check is now done against all the charts in the atlas. Since we store all the charts in the atlas, we remove the possibility of repeated coverage. However, as shown in [9] the possibility of early termination still exists. The effort is also unnecessarily expensive, as it does not rely on information about the distinction between boundary charts and interior charts.

4.2.2.3 Henderson's Method

In this section, we describe a method, given in [9], that addresses the problem of premature termination and the corresponding failure to cover all desired regions of the solution manifold. This algorithm is inspired by *Henderson's Algorithm* for continuation that we described in Section 2.3.2.

We associate each chart with a *convex polygon* P, having vertices $\bar{\omega}_i, i =$

$1, \ldots, k$ that lie in the corresponding tangent space. These points are ordered according to some default orientation of the polygonal boundary. We try to have P represent the domain of the solution manifold covered by its chart. Changes to the charts can be represented by a corresponding change in both the number of vertices as well as the corresponding direction vectors

$$s_i := \frac{\bar{\omega_i}}{||\bar{\omega_i}||} \tag{4.8}$$

and the normalized distances

$$v_i := ||\bar{\omega_i}|| \tag{4.9}$$

The expansion phase of the atlas algorithm that corresponds to the initialization of a new chart $\{u, T, \Sigma, R\}$ is now accompanied with the construction of a convex polygon P having n_{vx} vertices. This polygon is exterior to all points in the circle centered at u and having radius R. The direction vectors s_i are now given by:

$$s_i = \begin{pmatrix} \cos(2\pi(i-1)/n_{vx}) \\ \sin(2\pi(i-1)/n_{vx}) \end{pmatrix} \tag{4.10}$$

and

$$v_i = v := \frac{R}{\cos(\pi/n_{vx})} \tag{4.11}$$

for $i = 1, \ldots, n_{vx}$.

In the consolidation phase of the atlas algorithm, we *chop* the polygon P using straight lines in the corresponding tangent space that are determined by the intersection of the corresponding circle of radius R centered at ustar with the circle of radius R centerd at the orthogonal projection of the vector $u - u^*$, for some nearby base point u, onto the tangent space of u^* as given by

$$\varphi(u) := T \cdot T^T \cdot (u - u^*) \tag{4.12}$$

The polygon P associated with this chart is then replaced by the convex polygon

$$P\{r \in \mathcal{T}, ||r - \varphi(u^*)|| > ||r - \varphi(u)||\} \tag{4.13}$$

which is determined by subtracting the half-space of points which lie closer

to $\varphi(u)$ than to $\varphi(u^*) = 0$.

The set operation in the previous section is equivalent to removing all vertices that violate the inequality

$$r^T \cdot \varphi(u) \leq \frac{1}{2}||\varphi(u)||^2 \tag{4.14}$$

and then introducing new vertices along any edges of intersection of P with the line given by

$$r^T \cdot \varphi(u) = \frac{1}{2}||\varphi(u)||^2 \tag{4.15}$$

If, after some number of such set operations, there exists are least one polygonal vertex that lies outside a predefined radius $\tilde{R} < R$, that chart is said to be a *boundary chart*; otherwise it is known as an interior chart.

Our goal in obtaining a covering using such charts is to have the boundary of the manifold represented by a subset of the polygonal edges of the charts in the boundary array and to have the interior of the atlas covered by the interior charts. This can be achieved by always constructing new charts sufficiently close to existing ones. Note that every edge of the polygon that is created by the chopping operation can now be associated with the base point of the neighboring chart that initiated the chopping (and hence, generated that edge).

4.3 Parallel Atlas Algorithms

Covering higher dimensional surfaces presents us both a more complex problem and a unique opportunity: at every point in the solution manifold, no longer can our algorithm assume only two directions of further continuation, the possibility now exists of continuation in any direction in a plane containing the point. In higher dimensions, we acquire correspondingly greater degrees of freedom in choosing future continuation directions: the tangent space at a point u^*, $\mathcal{T}_{u^*}$ provides all such continuation directions. It is in this case that the availability of more processes to carry out continuation offers the chance of significant speedup. If, for instance, the base point and each of the N available continuation directions was given to a different worker process, then the solution manifold could presumably be computed N times faster.

However, such a theoretical speedup, if at all attainable, presents a serious bottleneck: once each of the several different workers have computed a part of the manifold, these parts must be merged together into one single manifold. This merging process must involve some kind of synchronization between the workers, so that the atlas that is being merged and stored represents the manifold accurately. The domain must be divided between the workers in a disjoint manner, to minimize duplication of effort. There must be some mechanism to prevent workers from selecting the same domain once it has been assigned to a worker. If this mechanism is too computationally expensive to implement, we might allow the same domain to be covered twice, but include a mechanism to handle the merging of two atlases when they both cover the same domain.

Let us try to mathematically describe the situation that we're discussing. We consider here a situation where one master process is doing the merging, and m worker processes are computing atlases over disjoint domains. We have a solution manifold M with a boundary array indexed by B, such that $M(B)$ gives us the list of charts on the boundary. Let t_a be the time required to compute a new atlas which covers a part of the solution manifold not covered by M; let t_{mi} be the time it takes to merge the i-th computed atlas into M. Suppose the manifold M is to be computed by the merger of n atlases. Then, the total computation time is given by:

$$T = \sum_{i=1}^{n} t_{mi} + t_a \cdot \frac{n}{m} \tag{4.16}$$

Let us consider the largest such merge time, $t_M = \max(t_{m1}, t_{m2}, \ldots t_{mn})$. Then,

$$T \leq n \cdot (t_M + \frac{t_a}{m}) \tag{4.17}$$

Here, it is assumed that $1 \leq m \leq n$. We see that by increasing m (i.e. increasing the number of parallel processes), we can reduce the proportion of total time spent computing new manifold; but t_M cannot be changed and hence can become a bottleneck. This is the main drawback of having a single process do the merging: the merge process quickly becomes the bottleneck and adding any number of extra processes won't change the computation time. (The astute reader might notice that what we have just described is a variation of the celebrated *Amdahl's Law*, which states that every program

has some minimum set of operations that have to be done sequentially [30]. Hence, while the time to perform other operations can be minimized by using parallelism, the program will never take less time to run than the time taken to run this minimum serial operation. In our case, the merging represents the inherently sequential part of the algorithm that cannot be parallelized.)

Now that we've recognized the bottleneck, we may wonder why it can't be eliminated. The simple answer is that the merge process requires having a consistent global manifold; hence only one process should be handling it. However, there are strategies where one can possibly divide the merge process among several workers as well, then there will need to be another process that just makes sure that computed domains do not overlap. While this may speed up the merge process, it would require more communication amongst the parallel processes. Thus, the bottleneck might now be in the time spent in communication, a situation often encountered when trying to solve a problem using parallel computation [31].

In the next section, we shall describe the first strategy that we used for designing parallel atlas algorithms in COCO and the reasons why it did not seem feasible. In Section 4.3.2 we describe an implementation of another strategy for solving the same problem; this is the strategy that we used in subsequent atlas algorithms. Finally, in Section 4.3.3, we describe a parallel atlas algorithm for covering 2D manifolds that offers the best chances of achieving faster computation times.

4.3.1 Parallel 1D Atlas Algorithm: Master-Worker in FSM

In this method, we make the parallel implementation transparent to the user. Both the master and the worker processes enter the FSM. The master remains in the `flush` state while the worker skips any computation in the flush state. Instead, the worker waits in the `predict` state for the master to send a boundary chart. Using this boundary chart, the worker computes the complete curve segment by iterating once through the `predict`, `correct` and `add` states. Once the complete curve segment is computed, this is passed to the master. The master then merges this curve segment into the atlas and proceeds to extract another boundary chart to send to a worker.

We show our code for the case of the 1D expanding boundary method

below:

Listing 4.1: Parallel 1D Covering: Expanding Boundary Method

```
function parallel_expanding_boundary()

stat = matlabpool('size');
if stat > 0
    matlabpool close;
end
matlabpool open local 3
spmd
run_code()
end
matlabpool close

end

function run_code()

prob = coco_add_func(coco_prob(), 'circle', @circle, [], ...
  'zero', 'u0', [1.5; 1]);
prob = coco_add_pars(prob, '', [1 2], {'x' 'y'});
prob = coco_set(prob, 'cont', 'PtMX', 100);
coco(prob, '1', [], 1, {'x' 'y'});

end
```

We start a collection of parallel MATLAB processes by using the **matlabpool** statement:

```
matlabpool open local 3
```

Here, the number at the end specifies the number of parallel processes to start. The statements before that:

```
stat = matlabpool('size');
if stat > 0
    matlabpool close;
end
```

ensure that any currently executing parallel processes are shut down using the **matlabpool** `close` statement. The construction of the continuation problem and the call to the `coco` entry point function are contained in a separate function:

```
function run_code()

prob = coco_add_func(coco_prob(), 'circle', @circle, [], ...
  'zero', 'u0', [1.5; 1]);
prob = coco_add_pars(prob, '', [1 2], {'x' 'y'});
```

```
prob = coco_set(prob, 'cont', 'PtMX', 100);
coco(prob, '1', [], 1, {'x' 'y'});

end
```

The above code simply initializes a continuation problem with a COCO-compatible function `circle` $f(u_1.u_2) = (u_1 - 1)^2 + u_2^2 - 1$:

Listing 4.2: coco-compatible function for a circle

```
function [data y] = circle(prob, data, u)

  y = (u(1)-1)^2+u(2)^2-1;

end
```

and an inital point $(1, 0)$.

This function is called from within the main function as:

```
spmd
run_code()
end
```

where **spmd** is a MATLAB statement that initializes environment variables that help in coding master-worker type of algorithms. For example, the `labindex` variable contains a unique id for each parallel process, and `numlabs` gives the number of parallel processes and initialized by **matlabpool**.

We now describe the changes made to the expanding boundary algorithm. Since we use a master-worker model, there must be logic for both a master and a worker process in the code for the FSM. There must also be a mechanism that selects the correct code for each process, i.e., the master and worker process should each only run their corresponding code. MATLAB's parallel computing toolbox provides the `labindex` variable; this variable indicates the unique ordinal number assigned of each worker process. We designate one such process as the master, and all the other processes as the worker. We also need to change the conditions that are checked to determine the next state; thus the master process will traverse the FSM differently than the worker.

Let us first see what changes are made to the subclass of `AtlasBase` used for the expanding boundary algorithm. The code for the original expanding boundary algorithm (described in [9]) is shown below:

Listing 4.3: `AtlasBase` **subclass used for the expanding boundary algorithm**

```
classdef atlas_1d_min < AtlasBase

  properties (Access=private)
    boundary = {};
    cont     = struct();
  end

  methods (Access=private)
    function atlas = atlas_1d_min(prob, cont, dim)
      assert(dim==1, '%s: wrong manifold dimension', mfilename);
      atlas      = atlas@AtlasBase(prob);
      atlas.cont = atlas.get_settings(cont);
    end
  end

  methods (Static)
    function [prob cont atlas] = create(prob, cont, dim)
      atlas = atlas_1d_min(prob, cont, dim);
      prob  = CurveSegment.add_prcond(prob, dim);
    end
  end

  methods (Static, Access=private)
    cont = get_settings(cont)
  end

  methods (Access=public)
    [prob atlas cseg correct] = init_prcond(atlas, prob, chart)
    [prob atlas cseg flush]   = init_atlas (atlas, prob, cseg)
    [prob atlas cseg]         = flush      (atlas, prob, cseg)
    [prob atlas cseg correct] = predict    (atlas, prob, cseg)
    [prob atlas cseg flush]   = add_chart  (atlas, prob, cseg)
  end

  methods (Access=private)
    flag = isneighbor(atlas, chart1, chart2)
    [atlas cseg] = merge(atlas, cseg)
  end

end
```

We see have seen an abstract form of such a subclass in listing 3.10. Here, the `boundary` property of the subclass is a cell array that holds at most two rows corresponding to the charts at the two endpoints of a 1D curve segment. We see that the class constructor `atlas_1d_min` includes a check for the dimensional deficit of the continuation problem:

```
assert(dim==1, '%s: wrong manifold dimension', mfilename);
```

There are two additional methods defined by this subclass: the `isneighbor`

method and the `merge` method. The former is used for checking whether two charts are neighbors of each other while the latter is used to merge a newly computed chart (stored in a `cseg` instance) into the atlas. Now we show the modified subclass instance that we use in our atlas algorithm:

Listing 4.4: Modified `AtlasBase` subclass used for our atlas algorithm

```
classdef atlas_1d_min < AtlasBase

  properties (Access=private)
    boundary    = {};
    cont        = struct();
    flagarray   = [];          % flagarray used in parallel comp
    next_pt     =  0 ;         % global counter for charts
  end

  methods (Access=private)
    function atlas = atlas_1d_min(prob, cont, dim)
      assert(dim==1, '%s: wrong manifold dimension', mfilename);
      atlas      = atlas@AtlasBase(prob);
      atlas.cont = atlas.get_settings(cont);
    end
  end

  methods (Static)
    function [prob cont atlas] = create(prob, cont, dim)
      atlas = atlas_1d_min(prob, cont, dim);
      prob  = CurveSegment.add_prcond(prob, dim);
    end
  end

  methods (Static, Access=private)
    cont = get_settings(cont)
  end

  methods (Access=public)
    [prob atlas cseg correct] = init_prcond(atlas, prob, chart)
    [prob atlas cseg flush]   = init_atlas (atlas, prob, cseg)
    [prob atlas cseg]         = flush      (atlas, prob, cseg)
    [prob atlas cseg correct] = predict    (atlas, prob, cseg)
    [prob atlas cseg flush]   = add_chart  (atlas, prob, cseg)
  end

  methods (Access=private)
    flag = isneighbor(atlas, chart1, chart2)
    [atlas cseg] = merge(atlas, cseg)
  end

end
```

The only change from the previous code is that here, we have two additional properties associated with this subclass: the `flagarray` property holds an

array of all the workers that are available, and the `next_pt` property holds the number of charts successfully merged by the `flush` function (described below). The master process does its most important work in the `flush` state. We first reproduce the complete code for the `flush` state in the expanding boundary algorithm below:

Listing 4.5: `flush` method of the expanding boundary algorithm

```
function [prob atlas cseg] = flush(atlas, prob, cseg)

if cseg.Status==cseg.CurveSegmentOK
  [atlas cseg] = atlas.merge(cseg);
end
[prob atlas cseg] = atlas.flush@AtlasBase(prob, cseg);
if cseg.Status==cseg.CurveSegmentOK
  if isempty(atlas.boundary) || ...
      (atlas.boundary{1,1}.pt>=atlas.cont.PtMX)
    cseg.Status = cseg.BoundaryPoint;
  end
end

end
```

This state has two main functions: to merge a chart into the atlas using the `merge` method and to check for termination of the atlas algorithm. The algorithm terminates either when the boundary cell array is empty of when the number of base points merged into the atlas is greater than the maximum limit given by `atlas.cont.PtMX`. We also observe that the flush method of the `AtlasBase` class is invoked.

The `flush` method, after our changes, is shown below:

Listing 4.6: `flush` method changed for parallelism

```
function [prob atlas cseg] = flush(atlas, prob, cseg)

if labindex == 1

  if cseg.isInitialSegment

    atlas.flagarray = ones(1, numlabs);

  else

    [cseg, src, tag] = labReceive();
    atlas.flagarray(src) = 1;
    chart    = cseg.curr_chart;
    atlas.next_pt = atlas.next_pt+1;
```

```
    chart.pt = atlas.next_pt+1;
    if chart.pt>=atlas.cont.PtMX
      chart.pt_type = 'EP';
      chart.ep_flag = 1;
    end

  end

  if cseg.Status==cseg.CurveSegmentOK
    [atlas cseg] = atlas.merge(cseg);
  end
  [prob atlas cseg] = atlas.flush@AtlasBase(prob, cseg);
  if cseg.Status==cseg.CurveSegmentOK
    if isempty(atlas.boundary) || ...
      (atlas.boundary{1,1}.pt>=atlas.cont.PtMX)
      cseg.Status = cseg.BoundaryPoint;
    end
  end
  % logic for scheduling job amongst workers
  [atlas.boundary, atlas.flagarray] = ...
    scheduleSend(atlas.boundary, ...
      cseg.Status, atlas.flagarray);
else % logic for the worker process

  if ~cseg.isInitialSegment
    labSend(cseg,1);
  end

  dat = labReceive(1);
  cseg.Status = dat{1}; % stop FSM
  atlas.boundary = dat{2};

end

end

function [boundary, farray] = ...
  scheduleSend(boundary, status, farray)

if  status == 0
  for i = 2:numlabs
    for j = 1:size(boundary, 1)
      if farray(i) & boundary{j,end}
        farray(i) = 0;
        boundary{j,end} = 0;
        labSend({status boundary(j, :)}, i);
        break
      end
    end
  end

else % terminate all labs

  pause(3);
  for i = 2:numlabs
    if ~farray(i)
```

```
      farray(i) = 1;
      dat = labReceive(i);
    end
    labSend({status []}, i);
  end

end

end
```

We observe that the `labindex` variable, provided by MATLAB's parallel computing environment, is used to decide between which code to run on the master process `labindex == 1` and which to run on the worker process `labindex > 1`.

Here, we check whether this is the first run through the FSM by observing the value of the `cseg.isInitialSegment` variable. If that is the case, we initialize an array that indicates which workers are busy and which are free.

If this is not the first run, then the master receives a structure that encodes the curve segment (the `cseg` structure) and merges it into the solution manifold using the `merge` function.

Finally, the master calls the `scheduleSend` function which is shown below:

```
function [boundary, farray] = ...
  scheduleSend(boundary, status, farray)

if  status == 0
  for i = 2:numlabs
    for j = 1:size(boundary, 1)
      if farray(i) & boundary{j,end}
        farray(i) = 0;
        boundary{j,end} = 0;
        labSend({status boundary(j, :)}, i);
        break
      end
    end
  end

else % terminate all labs

  pause(3);
  for i = 2:numlabs
    if ~farray(i)
      farray(i) = 1;
      dat = labReceive(i);
    end
    labSend({status []}, i);
  end
```

```
end

end
```

This code contains logic for terminating all worker processes once the computation is deemed to be complete. It does so by sending a nonzero value as the first input parameter to the structure passed by the `labSend` function. This is an indication to the worker that it should terminate by exiting the FSM.

The code also deduces which worker is free and sends a boundary chart to the free worker. The worker will use this as the base chart to construct a curve segment:

```
if ~ cseg.isInitialSegment
    labSend(cseg,1);
end
dat = labReceive(1);
cseg.Status = dat{1}; % stop FSM
atlas.boundary = dat{2};
```

Here, the worker gets a boundary chart from the master and stores it in the `boundary` property of the `atlas` structure. This is then used in the `predict` state to construct a new curve segment for further continuation.

The encoding for the `predict` class method in the expanding boundary algorithm from [9] is given below:

Listing 4.7: `predict` method of the expanding boundary algorithm

```
function [prob atlas cseg correct] = predict(atlas, prob, cseg)

[chart xp s h] = atlas.boundary{1,:};
prcond = struct('x', chart.x, 'TS', chart.TS, 's', s, 'h', h);
th             = atlas.cont.theta;
if th>=0.5 && th<=1
  xp           = chart.x+(th*h)*(chart.TS*s);
  [prob cseg]  = CurveSegment.create(prob, chart, prcond, xp);
  [prob ch2]   = cseg.update_TS(prob, cseg.curr_chart);
  h            = h*(ch2.TS'*chart.TS);
  xp           = chart.x+h*(ch2.TS*s);
  prcond = struct('x', chart.x, 'TS', ch2.TS, 's', s, 'h', h);
end
[prob cseg] = CurveSegment.create(prob, chart, prcond, xp);
correct     = true;

end
```

We see that a chart is extracted from the boundary array and then used as an

input to the `Curvesegment.create` method, which initializes a partial curve segment and a corrector.

The changes that we make in our code is shown below:

Listing 4.8: `predict` method with parallelism enabled

```
function [prob atlas cseg correct] = predict(atlas, prob, cseg)

if labindex > 1 % logic for the worker process
  [chart xp s h] = atlas.boundary{1,:};
  prcond = struct('x', chart.x, 'TS', chart.TS, 's', s, 'h', h);
  th             = atlas.cont.theta;
  if th>=0.5 && th<=1
    xp           = chart.x+(th*h)*(chart.TS*s);
    [prob cseg] = CurveSegment.create(prob, chart, prcond, xp);
    [prob ch2]  = cseg.update_TS(prob, cseg.curr_chart);
    h            = h*(ch2.TS'*chart.TS);
    xp           = chart.x+h*(ch2.TS*s);
    prcond       = ...
      struct('x', chart.x, 'TS', ch2.TS, 's', s, 'h', h);
  end
  [prob cseg] = CurveSegment.create(prob, chart, prcond, xp);
  correct      = true;
else % logic for the master process
  [prob cseg]   = ...
    CurveSegment.create_initial(prob, atlas.boundary{1,1}, false);
  correct      = false;

end

end
```

As before, the check on the `labindex` variable ensures that master and worker processes run different parts of the code even though they use the same file. In this case, the worker runs the exact same code as described in the original algorithm (listing 4.7). In order to ensure compatibility with the core implementation of the FSM, the master process does compute a curve segment, but it is not used for continuation. The output boolean argument `correct` is set to false; therefore, the master does not enter the `correct` state but transitions into the `add_chart` state.

We next show the encoding of the `add_chart` method in the expanding boundary algorithm given in [9]:

Listing 4.9: `add_chart` method of the expanding boundary algorithm

```
function [prob atlas cseg flush] = add_chart(atlas, prob, cseg)
```

```
chart    = cseg.curr_chart;
chart.pt = chart.pt+1;
if chart.pt>=atlas.cont.PtMX
  chart.pt_type = 'EP';
  chart.ep_flag = 1;
end
[prob cseg] = cseg.add_chart(prob, chart);
flush       = true;

if ~atlas.isneighbor(cseg.ptlist{1}, cseg.ptlist{end})
  cseg.ptlist{end}.pt_type = 'GAP';
  cseg.ptlist{end}.ep_flag = 2;
  cseg.Status              = cseg.CurveSegmentCorrupted;
end

end
```

This method reads the most recently located point and adds it to a list of charts on the manifold, stored in the `cseg` instance.

The changes that we make to enable parallellism are shown below:

Listing 4.10: `add_chart` method with parallelism enabled

```
function [prob atlas cseg flush] = ...
  add_chart(atlas, prob, cseg)

if labindex > 1

  chart    = cseg.curr_chart;
  chart.pt = chart.pt+1;
  if chart.pt>=atlas.cont.PtMX
    chart.pt_type = 'EP';
    chart.ep_flag = 1;
  end
  [prob cseg] = cseg.add_chart(prob, chart);
  flush       = true;

  if ~atlas.isneighbor(cseg.ptlist{1}, cseg.ptlist{end})
    cseg.ptlist{end}.pt_type = 'GAP';
    cseg.ptlist{end}.ep_flag = 2;
    cseg.Status     = cseg.CurveSegmentCorrupted;
  end

else

  cseg.Status = ~CurveSegmentBase.CurveSegmentOK;
  cseg.isInitialSegment = false;
  flush       = true;

end

end
```

Again, we see that the worker runs the exact same code as before; the only difference is in the code for the master. Here, we see that the `isInitialSegment` flag of the `cseg` instance is set to `false`; this boolean flag indicates whether the master is traversing the FSM for the first time. The `cseg.Status` flag is set such that the master does not transition from the `flush` to the `exit` state in the first iteration, but continues in a loop around the FSM.

Next, we show the `merge` function of the expanding boundary algorithm, as given in [9]:

Listing 4.11: `merge` method of the expanding boundary algorithm

```
function [atlas cseg] = merge(atlas, cseg)

chart = cseg.ptlist{end};
R     = atlas.cont.h;
h     = atlas.cont.Rmarg*R;
nb    = cell(2,4);
for k=1:2
  sk      = chart.s(k);
  xk      = chart.x+h*(chart.TS*sk);
  nb(k,:) = {chart, xk, sk, h};
end
for i=size(atlas.boundary,1):-1:1
  chart2 = atlas.boundary{i,1};
  if atlas.isneighbor(chart, chart2)
    x2 = atlas.boundary{i,2};
    if norm(chart.TS'*(x2-chart.x))<R
      atlas.boundary(i,:) = [];
    end
    for k=size(nb,1):-1:1
      x1 = nb{k,2};
      if norm(chart2.TS'*(x1-chart2.x))<R
        nb(k,:) = [];
      end
    end
  end
end
atlas.boundary = [nb; atlas.boundary];
if isempty(atlas.boundary)
  chart.pt_type    = 'EP';
  chart.ep_flag    = 1;
  cseg.ptlist{end} = chart;
end

end
```

This method implements the conditions that were described in 4.2.1.3 to check whether the most recently located point on the curve segment by the atlas algorithm lies close enough to the boundary of the computed atlas. We

make two small changes to the above method:

```
nb = cell(2,5);
```

and

```
nb(k,:) = {chart, xk, sk, h, 1};
```

These changes correspond to adding one more element to the arrays stored in the boundary array `atlas.boundary`. This element is used later by the master process to ensure that no two worker process are given the same boundary chart.

Next, we show the classmethods associated with the `init` state that were changed to enable parallelism. We made the following changes to the `init_prcond` method:

Listing 4.12: `init_prcond` method of the expanding boundary algorithm

```
function [prob atlas cseg correct] = init_prcond(atlas, prob, chart)

chart.R       = 0;
chart.pt      = -1;
chart.pt_type = 'IP';
chart.ep_flag = 1;
[prob cseg]   = CurveSegment.create_initial(prob, chart);
correct       = cseg.correct;

end
```

This method initializes a curvesegment and optionally enters a correct state.

In our modification, since only the master computes the initial curve segment, we allow the workers to skip the `correct` state as shown below:

Listing 4.13: `init_prcond` method with parallelism enabled

```
function [prob atlas cseg correct] = init_prcond(atlas, prob, chart)

chart.R       = 0;
chart.pt      = -1;
chart.pt_type = 'IP';
chart.ep_flag = 1;
[prob cseg]   = CurveSegment.create_initial(prob, chart);

if labindex == 1
    correct       = cseg.correct;
else
    correct = false;
end
```

```
end
```

The rest of the methods remain the same as those from the expanding boundary algorithm.

In our implementation, it can be seen that no changes are required to any code in the problem construction; that code remains the same (we do need to add parallel computing primitives like **`matlabpool`** here, but the fact is that the construction of the continuation problem structure `prob` and the call to `coco` entry point function did not change). This was by design: we wanted to make the parallel computing part of the code as transparent to the user as possible.

Numerical experiments with this implementation showed that, for problems with relatively few unknowns, $t_a << t_m$, where t_a is the time taken by a worker to compute a new curve segment and t_m is the time taken by the master to merge this curve segment into the atlas. What this meant was that the worker process was computing very very fast; and hence the proportion of time taken by the master to merge the curve segment dominated the overall runtime. But since this is a serial operation (only the master can merge), the speed of the computation could not be made any faster.

It is for this reason that we looked for an alternative strategy for parallel computing. Our only option was to have more work done by the parallel workers; then the proportion of total runtime taken up by the master would be less. The actual strategy that we finally employed is detailed in the following sections.

4.3.2 Removing the Master process from the FSM

In the last section, we have seen that having only the master process undertake the process of merging does not give us much benefit as the bottleneck in the computation is found to be the merge process by the master.

One simple idea to get better performance using parallelism is thus to have the workers do more work. In particular, we consider the idea of workers computing a *local atlas* of a few limited charts around their initial solution point. If every worker computes a local atlas over its domain, and we divide the domain amongst the workers in a disjoint way, then this patchwork of

local atlases can be said to approximate the covering of the manifold. The task of merging these local atlases is now handled by the master process.

Assuming that we use such a method, we see that the master does not need to run the continuation algorithm; it simply has to decide which initial base point to give to the workers, and then wait for the worker to compute a local atlas. The master then merges the local atlas into the global atlas. The next base point to give to the worker is now obtained from the boundary of the global atlas. Thus, at every stage, progress is being made towards the goal of obtaining a covering of the solution manifold.

We may wonder why this global merging cannot be handled by the worker processes themselves. Although it is possible to do so, it would require a lot of communication between the processes. First, let us consider what such a process would mean, when working with a distributed memory architecture. There is no global memory which all the workers can access; hence any representation of a global structure must be a structure with a local copy in each of the workers. However, since the global atlas must be *globally consistent* (i.e., must have the same value when accessed by any process), if a process makes changes to the atlas, it must communicate its change to *every* other process, who must then incorporate those changes into their respective versions of the atlas. This communication overhead quickly nullifies any percieved benefit we might obtain from using a distributed version of merging.

We illustrate such a scheme in the case of 1D atlas algorithms. However, we are only showing how the master can send a continuation problem structure to the worker; the part where the worker sends back the results in not shown. We will implement the full idea in the case of higher dimensional atlas algorithms in the next section.

```
stat = matlabpool('size');
if stat > 0
    matlabpool close;
end

matlabpool open local 2

spmd

if labindex == 1
    % create the problem structure
    prob = coco_add_func(coco_prob(), 'circle',...
      @circle, [], 'zero', 'u0', [1.5; 1]);
    prob = coco_set(prob, 'cont', 'PtMX', 100);
    labSend({prob}, 2);
```

```
    labRecieve();
else
    data = labReceive(1);
    prob = data{1}
    coco(prob, '1', [], 1);
    labSend('computation complete', 1);
end

end

matlabpool close
```

Note that here, the master does not receive the data back from the worker. We use this code only to illustrate the idea of separating master and worker code during the problem construction stage, instead of in the FSM. We shall use this concept in the computation of 2D manifolds as discussed below.

4.3.3 Parallel Covering of 2D Manifold using COCO

When attempting to achieve faster computation times (or *speedup*) through parallel computation, there are two fundamental problems that must be resolved efficiently:

1. How the domain is to be divided among the workers;
2. How the computed partial result is to be combined into one complete solution.

We shall discuss each of these issues in detail in this section. The atlas algorithm that we use as a base is *Henderson's* atlas algorithm that was described in Section 4.2.2.3. Accordingly, the charts that we use will each correspond to a convex polygon P with vertices labelled $\bar{\omega}_i, i = 1, \ldots, k$ which lie in the tangent space of the base point u^*.

4.3.3.1 Merging Atlases

We consider first the problem of merging a partial atlas constructed by a worker with the global atlas maintained by master. We imagine in this case, that individual workers implement both the expansion and consolidation phases of manifold covering, but do so within a computational domain that is defined by the master process.

Let us first show the subclass of the `AtlasBase` class that is used in Henderson's atlas algorithm, as given in [9].

Listing 4.14: `AtlasBase` subclass used for Henderson's atlas algorithm

```
classdef atlas_2d_min < AtlasBase

  properties (Access=private)
    boundary = [];
    charts   = {};
    next_pt  = 0;
    cont     = struct();
  end

  methods (Access=private)
    function atlas = atlas_2d_min(prob, cont, dim)
      assert(dim==2, '%s: wrong manifold dimension', mfilename);
      atlas      = atlas@AtlasBase(prob);
      atlas.cont = atlas.get_settings(cont);
    end
  end

  methods (Static)
    function [prob cont atlas] = create(prob, cont, dim)
      atlas = atlas_2d_min(prob, cont, dim);
      prob  = CurveSegment.add_prcond(prob, dim);
      prob  = coco_add_slot(prob, 'atlas', ...
        @atlas.save_atlas, [], 'save_bd');
    end

    function [data res] = save_atlas(prob, data, varargin)
      res.charts   = prob.atlas.charts;
      res.boundary = prob.atlas.boundary;
    end
  end

  methods (Static, Access=private)
    cont = get_settings(cont)
  end

  methods (Access=public)
    [prob atlas cseg correct] = init_prcond(atlas, prob, chart)
    [prob atlas cseg flush]   = init_atlas (atlas, prob, cseg)
    [prob atlas cseg]         = flush      (atlas, prob, cseg)
    [prob atlas cseg correct] = predict    (atlas, prob, cseg)
    [prob atlas cseg flush]   = add_chart  (atlas, prob, cseg)
  end

  methods (Access=private)
    flag = isneighbor(atlas, chart1, chart2)
    flag = isclose(atlas, chart1, chart2)
    [atlas cseg] = merge(atlas, cseg)
    [atlas chart1 checked] =
      merge_recursive(atlas, chart1, k, checked)
    chart =
```

```
        subtract_half_space(atlas, chart, test, phi, flag, NB)
    end
```

The `charts` property is a cell array that contains all the charts constructed by the atlas algorithm. The `boundary` array is an array of ordinal numbers that indexes into `charts` to give a cell array of charts that are on the boundary of the manifold. The `save_atlas` method stored the cell array of charts and the boundary array in its `res` output argument; COCO then stores the `res` structure to disk.

Careful analysis shows that the default handling of computational boundaries by the atlas algorithms given in [9] is insufficient for the purpose of allowing the master process to merge a partial atlas constructed by a worker into the global atlas. In particular, charts on the boundary of the computational domain are removed from the `boundary` array and then considered to be equivalent to interior charts. These charts would therefore not be used for further continuation. However, since we impose restrictions on the computational domain as a way of distributing the computation among multiple workers, we must now enable a mechanism to track the charts on the boundary of the computational domain; otherwise it would not be possible to merge the results of the workers into the global atlas.

We add a property called `compbound` to the atlas subclass. This is an array of ordinal numbers that stores the charts that are on the boundary of the computational domain. To keep track of the computational domain, the `boundary` is merged with this array before storing it to disk:

```
res.boundary = union(prob.atlas.boundary,prob.atlas.compboun);
```

To merge a newly computed chart into the atlas, a recursive merge algorithm is described in [9]. The implementation of this algorithm is reproduced below:

Listing 4.15: `merge_recursive` method of Henderson's algorithm

```
function [atlas chart1 checked] = ...
  merge_recursive(atlas, chart1, k, checked)

checked(end+1) = k;
chartk = atlas.charts{k};
if atlas.isclose(chart1, chartk)
dx     = chartk.x-chart1.x;
phi1   = chart1.TS'*dx;
```

```
  phik   = chartk.TS'*(-dx);
  test1  = chart1.v.*(chart1.s'*phi1)-norm(phi1)^2/2;
  testk  = chartk.v.*(chartk.s'*phik)-norm(phik)^2/2;
  flag1  = (test1>0);
  flagk  = (testk>0);
  chart1 = ...
    atlas.subtract_half_space(chart1, test1, phi1, ...
      flag1, chartk.id);
  chartk = ...
    atlas.subtract_half_space(chartk, testk, phik, ...
      flagk, chart1.id);
  atlas.charts{k} = chartk;
  check = setdiff(chartk.nb, checked);
  while ~isempty(check)
    [atlas chart1 checked] = ...
      atlas.merge_recursive(chart1, check(1), checked);
    check = setdiff(chartk.nb, checked);
  end
  end

  end
```

The input arguments of this function consists of an `atlas`, a chart, `chart1`; an integer `k` and an array `checked`. The `k` input integer gives the ordinal number of the chart from the atlas that is to be merged with the input chart, `chart1`. The `checked` input argument is an array of ordinal numbers that indicates all the charts which have already been merged by this method (and hence must not be merged again).

This method first extracts the chart indexed by `k` into `chartk`. It then checks if `chartk` overlaps `chart1` by invoking the `isclose` function. If they do, both charts are *chopped* to remove their regions of overlap. The chopping process was described in section 4.2.2.3; and is implemented by the `subtract_half_space` method. Once both the charts are chopped, the method stores `chartk` back in the atlas. It also stores the list of all neighbors of `chartk` that have not been checked before in a new array called `check`. The algorithm then calls itself recursively until all its neighbors have been checked. This recursive algorithm is guaranteed to terminate due to the presence of the `checked` and `check` array (the `check` array is given as the `checked` input argument in the recursive call): every call to the `merge_recursive` method adds at least one element to the `checked` array. Hence, there can only be n calls to `merge_recursive`, where n is the total number of charts in the atlas.

In our case, we are required, not to merge a chart, but the entire *atlas*. An extension of the strategy described for a single chart seems appropriate. We

present such an implementation below:

Listing 4.16: a recursive merge of two atlases

```
function atlas = atlas_merge(atlas1, atlas2, cont)

% merge 2 atlas, atlas1 and atlas2, store result in
% atlas

% the atlas with greater number of charts is
% assigned to atlas

if atlas1.charts{end}.id>=atlas2.charts{end}.id
  atlas = atlas1;
else
  atlas = atlas2;
  atlas2 = atlas1;
end
size = atlas.charts{end}.id;

% store pairs of coincident charts in a cell
% array

atlas.coincident = {};

neighbors = atlas2.boundary;
bd_charts = atlas.boundary;

% function that merges charts of atlas and atlas2
% charts in both atlases are changed, but no charts
% are moved from one atlas to another

[atlas, atlas2] = ...
merge_into(size, neighbors, bd_charts, atlas, atlas2, cont);

% for all charts of atlas2 that were not changed
% by the merge_into function, update the id's of both
% the charts and the ordinal numbers stored in their
% nb fields(as they refer to charts whose id has been
% changed)

  for i=1:numel(atlas2.charts)
    if ~isfield(atlas2.charts{i},'check')
      atlas2.charts{i}.id = atlas2.charts{i}.id+size;
      atlas2.charts{i}.nb(atlas2.charts{i}.nb>0) = ...
        atlas2.charts{i}.nb(atlas2.charts{i}.nb>0)+size;
    end
  end

% add the charts from atlas2 into atlas

atlas.charts   = [atlas.charts, atlas2.charts];

% merge the boundary array
```

```
atlas.boundary = union(atlas.boundary, atlas2.boundary+size);

% update the boundary array so that only charts with
% nonempty bv are referenced by the boundary array

bd_charts      = atlas.charts(atlas.boundary);
idx            = cellfun(@(x) ~isempty(x.bv), bd_charts);
atlas.boundary = atlas.boundary(idx);

% now handle any charts that are coincident

if ~isempty(atlas.coincident)
  for i=1:numel(atlas.coincident)
    pair = atlas.coincident{i};
    nbhd = atlas.charts{pair(1)}.nb
    for j=1:numel(nbhd);
      if nbhd(j) == 0
        continue
      end
      atlas.charts{nbhd(j)}.nb(atlas.charts{nbhd(j)}.nb ...
        == pair(1)) = pair(2);
    end

    % remove one chart of the pair from the atlas

    atlas.charts{pair(1)} = [];
    atlas.boundary = setdiff(atlas.boundary, pair(1));
    atlas = close_atlas(atlas, cont, pair(2));
  end
end
atlas = rmfield(atlas, 'coincident');

% function for double-recursive merge of two atlases
function [atlas, atlas2] = ...
  merge_into(size, neighbors, ...
    bd_charts, atlas, atlas2, cont)

% input arguments
% atlas:     an atlas
% atlas2:    another atlas
% size:      id of the last chart of atlas
% cont:      structure with predefined settings
% neighbors: boundary charts of atlas2
% bd_charts: boundary charts of atlas

  for i=1:numel(neighbors)
    chart        = atlas2.charts{neighbors(i)};

    % update the id field of the chart if not
    % done so already

    if ~isfield(chart,'check')
        chart.id = chart.id+size;
        chart.nb(chart.nb>0) = chart.nb(chart.nb>0)+size;
        chart.check = [];
    end
```

```
      % find all charts in atlas2 that are close to the
      % boundary charts of atlas and have not already
      % been modified

      nbfunc       = @(x) isclose(chart, x, cont);
      idx          = cellfun(nbfunc, atlas.charts(bd_charts));
      close_charts = setdiff(bd_charts(idx), chart.check);
      if ~isempty(close_charts)
        checked    = [0, chart.id];

        % merge one chart from close_chart with the
        % charts in atlas2

        while ~isempty(close_charts)
          [atlas, chart, checked] = ...
            merge_recursive(size, atlas, ...
              chart, close_charts(1), checked, cont);
            close_charts = setdiff(close_charts, checked);
        end
        chart.check = [chart.check, checked];
        atlas2.charts{neighbors(i)} = chart;

        % recursively select all charts in atlas and merge
        % until no chart in atlas2 is close
        % and not checked

        [atlas, atlas2] = ...
          merge_into(size, chart.nb(chart.nb>size)-size, ...
            bd_charts, atlas, atlas2, cont);
      end
    end
  end
  end
```

The `atlas_merge` function takes as input two atlases `atlas1` and `atlas2`. The other input argument is a `cont` structure that holds some predefined settings.

We now discuss the salient parts of this code. In particular, the `merge_into` function. In this function, the charts from both atlases are changed so that they do not overlap with one another. Its input arguments are:

- `atlas`: an atlas;
- `atlas2`: another atlas;
- `size`: `id` of the last chart in `atlas`;
- `bd_charts`: boundary charts of `atlas`;

- `neighbors`: charts of `atlas2` that need to be checked for overlap with charts in `atlas`;
- `cont`: structure that contains predefined settings

The function iteratively selects a chart from the `neighbors` input argument and merges it into `atlas`, if there's overlap. The recursive merge of a chart from `atlas2` into `atlas` is almost identical to the `merge_recursive` method that we described in listing 4.15 (there are a few differences which will be discussed below). Once a chart from the `neighbors` input argument is merged, the function calls itself with the same input arguments *except* the `neighbors` argument, which is now set to represent all its neighboring charts in `atlas2`. Hence, the algorithm first checks the boundary charts of `atlas2` for overlap; whenever such a chart is detected, it is merged with `atlas2`. Then, its neighbors in `atlas2`, that have not already been merged, are now recursively merged with `atlas`.

It might happen that two charts are found to be coincident. This is detected by the function below:

Listing 4.17: Determine if two charts are coincident

```
function flag = iscoincident(chart1, chart2)

x1   = chart1.x;
x2   = chart2.x;
dx   = x2-x1;
phi1 = chart1.TS'*dx;
phi2 = chart2.TS'*dx;
x1s  = chart1.TS*(phi1);
x2s  = chart2.TS*(phi2);
dst  = [norm(x1s), norm(x2s), norm(dx-x1s), norm(dx-x2s), ...
        subspace(chart1.TS, chart2.TS)];
dstmx = [1e-10, 1e-10, 1e-10, 1e-10, 1e-10];
flag  = false;
if all(dst<dstmx)
    flag = true;
end

end
```

In this function, the same criteria that was used in Eq. (4.7) to compute if two charts overlap is now used to check whether the two charts are coincident.

The check for coincidence between charts is done in the `merge_recursive` function that is given below:

Listing 4.18: Recursive merge of a chart with an atlas

```
function [atlas, chart1, checked] = ...
    merge_recursive(size, atlas, chart1, k, checked, cont)
checked(end+1) = k;
chartk = atlas.charts{k};
if isclose(chart1, chartk, cont)
  dx     = chartk.x-chart1.x;
  phi1   = chart1.TS'*dx;
  phik   = chartk.TS'*(-dx);
  test1  = chart1.v.*(chart1.s'*phi1)-norm(phi1)^2/2;
  testk  = chartk.v.*(chartk.s'*phik)-norm(phik)^2/2;
  flag1  = (test1>1e-10);
  flagk  = (testk>1e-10);
  if ~isempty(find(flag1, 1))
  chart1 = ...
    subtract_half_space(chart1, test1, ...
      phi1, flag1, chartk.id, cont);
  end
  if ~isempty(find(flagk, 1))
    chartk = ...
      subtract_half_space(chartk, testk, ...
        phik, flagk, chart1.id, cont);
  end
  atlas.charts{k} = chartk;
  check = setdiff(chartk.nb(chartk.nb<=size), checked);
  while ~isempty(check)
      [atlas, chart1, checked] = ...
          merge_recursive(size, atlas, ...
            chart1, check(1), checked, cont);
      check = setdiff(chartk.nb(chartk.nb<=size), checked);
  end
elseif iscoincident(chart1, chartk)
  atlas.coincident = [atlas.coincident, [chart1.id, k]];
end

end
```

This function is based on the class method given in listing 4.15. However, we see two important differences:

- before a chart is chopped, we check whether at least one of the conditions for overlap are satisfied;
- if two charts are not determined to be overlapping by the `isclose` function, they are checked for coincidence.

In case two charts are coincident, they are stored in a property of the `atlas` structure called `coincident`. This is a cell array that contains pairs of coincident charts as its elements. We are concerned about checking for coincident

charts because they are not handled correctly by the `subtract_half_space` function, given in listing A.14. Since coincident charts do not intersect, we end up duplicating charts, which is highly undesirable when we want a compatible covering.

After the two atlases are merged and the combined atlas is stored in `atlas`, the algorithm checks for coincident charts as shown below:

```
if ~isempty(atlas.coincident)
  for i=1:numel(atlas.coincident)
    pair = atlas.coincident{i};
    nbhd = atlas.charts{pair(1)}.nb
    for j=1:numel(nbhd);
    if nbhd(j) == 0
      continue
    end
    atlas.charts{nbhd(j)}.nb(atlas.charts{nbhd(j)}.nb ...
      == pair(1)) = pair(2);
    end

    % remove one chart of the pair from the atlas

    atlas.charts{pair(1)} = [];
    atlas.boundary = setdiff(atlas.boundary, pair(1));
    atlas = close_atlas(atlas, cont, pair(2));
  end
end
atlas = rmfield(atlas, 'coincident');
```

In the above algorithm, every pair of coincident charts is considered. The neighbors of each chart in every pair are changed accordingly, and one of the charts is removed from the atlas. Once the chart is removed, the `close_atlas` function is called, which is given below:

Listing 4.19: Function to ensure consistency of atlas after chart removal

```
function atlas = close_atlas(atlas, cont, k)

chart        = atlas.charts{k};
chart.v      = chart.R*ones(numel(chart.v),1);
close_charts = chart.nb;
checked      = [0, chart.id];
while ~isempty(close_charts)
[atlas, chart, checked] = ...
    merge_recursive(atlas.charts{end}.id, atlas, ...
      chart, close_charts(1), checked, cont);
close_charts = setdiff(close_charts, checked);
end
atlas.charts{k} = chart;
if isempty(chart.bv)
```

```
    atlas.boundary = setdiff(atlas.boundary, 1);
end

end
```

This function uses the `merge_recursive` function with the chart that is kept. This ensures that all its neighboring charts stay consistent after the removal of a chart from the atlas.

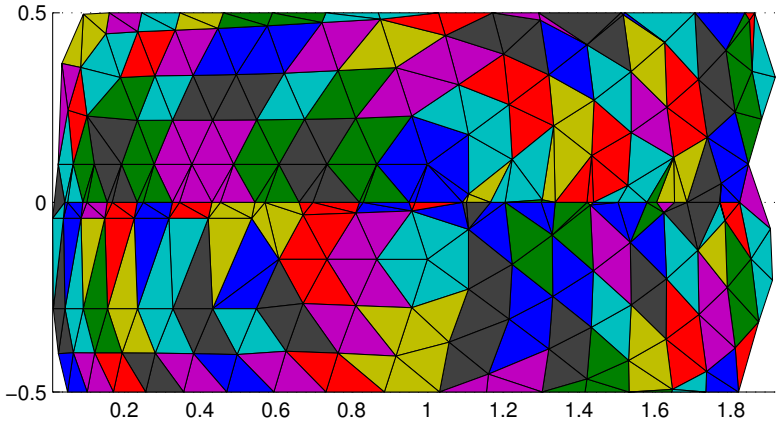
Finally, we note that the `isclose` function used to determine overlap between two charts is the same as `isclose` class method described in [9]. We reproduce it here for completeness:

Listing 4.20: function to determine if two charts are close to each other

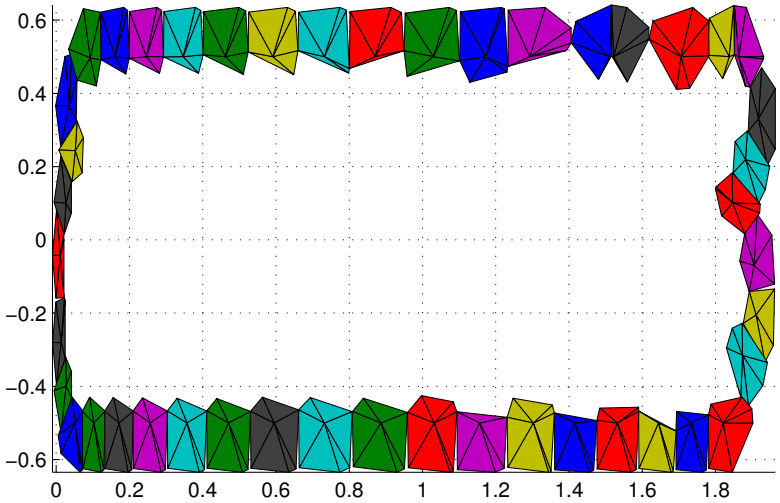
```
function flag = isclose(chart1, chart2, cont)

% k contains the minimum dimension of the base points
k    = cont.mind;
al   = cont.almax;
R    = cont.h;
ta   = tan(al);
t2a  = tan(2*al);
x1   = chart1.x(1:k);
x2   = chart2.x(1:k);
dx   = x2-x1;
phi1 = chart1.TS(1:k,:)'*dx;
phi2 = chart2.TS(1:k,:)'*dx;
x1s  = chart1.TS(1:k,:)*(phi1);
x2s  = chart2.TS(1:k,:)*(phi2);
dst  = [norm(x1s), norm(x2s), norm(dx-x1s), norm(dx-x2s), ...
        subspace(chart1.TS(1:k,:), chart2.TS(1:k,:))];
n1mx  = ta*min(R,norm(x1s))+t2a*max(0,norm(x1s)-R);
n2mx  = ta*min(R,norm(x2s))+t2a*max(0,norm(x2s)-R);
dstmx = [2*R, 2*R, n1mx, n2mx, 2*al];
flag  = false;
if all(dst<dstmx);
  test1 = chart1.v.*(chart1.s'*phi1)-norm(phi1)^2/2;
  test2 = chart2.v.*(chart2.s'*phi2)+norm(phi2)^2/2;
  flag  = any(test1>0) && any(test2<=1e-10);
end

end
```

The results of this recursive merge can be seen in Figure 4.1a and 4.1b when merging two cylinders along their boundary at $z = 0$.

One of the major difficulties with this approach is the problem of merging charts that are the result of the solution of different continuation problems. This situation is likely to happen in our algorithm as we specify the domain



(a) All charts of the merged atlas



(b) Boundary charts of the merged atlas

Figure 4.1: Merging two cylinders along their boundary; the boundary of the two cylinders is the $z = 0$ plane. Recursive merging changes charts in both the atlases to be compatible with each other

boundaries by adding monitor functions and then restricting the values of the continuation parameters so as to restrict the solution manifold to the required domain. However, the addition of parameters to the problem also increases the dimension of the solution space itself. Therefore, the merge algorithm, which was initially developed for merging charts representing solution spaces of the same dimension, has to be changed to handle solution spaces of different dimensions.

Our solution to this problem has been to use a minimum dimension in our algorithm when merging charts computed from different problems. Since all the charts are more constrained versions of the original problem, they all have at least as many parameters as in the original problem itself. Thus, we use this as the minimum dimension across which to perform comparisons and merges. The implementation of this idea can be seen in the code for the master process in listing A.10. We reproduce the snippet of interest below:

```
...
...
% create the problem structure

prob  = create_prob(u0, {}, {});
prob = coco_add_func(coco_prob(), 'cylinder', ...
  @cylinder, [], 'zero', 'u0', u0);
% store the minimum dimension in cont
cont.mind = prob.efunc.x_dim;

prob = coco_add_pars(prob, '', [1, 2, 3], ...
  {'x' 'y' 'z'});
...
...
```

Here, the `cont.mind` attribute is assigned the minimum dimension of the zero problem. This value is used in the `isclose` function to determine overlap between charts corresponding to base points of different dimensions (as shown in listing 4.20)

4.3.3.2 Domain Decomposition

We proceed to describe the process of domain decomposition, the division of the computational domain allocated to each worker. However, it should be noted that this term can have many meanings; in [32], at least three different meanings are attributed to this term. In parallel computing, domain decomposition refers to the process of dividing the data from a computational

model amongst the processors in a *distributed memory architecture* (defined in Section 2.4.3). In the case of manifold covering, we seek an algorithm for the process of domain decomposition that is independent of the problem being solved.

Most extant literature deals with the problem of decomposing a domain which is known *a priori*. In our case, the domain of interest (the manifold of solutions) is not available to us *a priori*.

In the absence of global information about the problem domain, we propose the following heuristic solution. Consider a boundary chart of the atlas. We determine a vertex v of the polygon that is farthest from the interior. The domain is now split using a plane orthogonal to the vector passing through the center of the chart and point v, as illustrated in Figure 4.2.

We now describe how we implemented such a domain decomposition. Since we already have a powerful continuation platform with facilities to extend the problem and then restrict it by constraining the added parameters, it was natural to think of using COCO itself to decompose the domain. Here, we admit that we were quite lucky; domain decomposition is usually a hard problem. It requires significant effort to both decide the kind of domains into which the problem must be divided and to implement such a scheme into an existing code that solves the problem.

The equation of a plane passing through a point u_0 and having a normal $\hat{n}$ is given by:

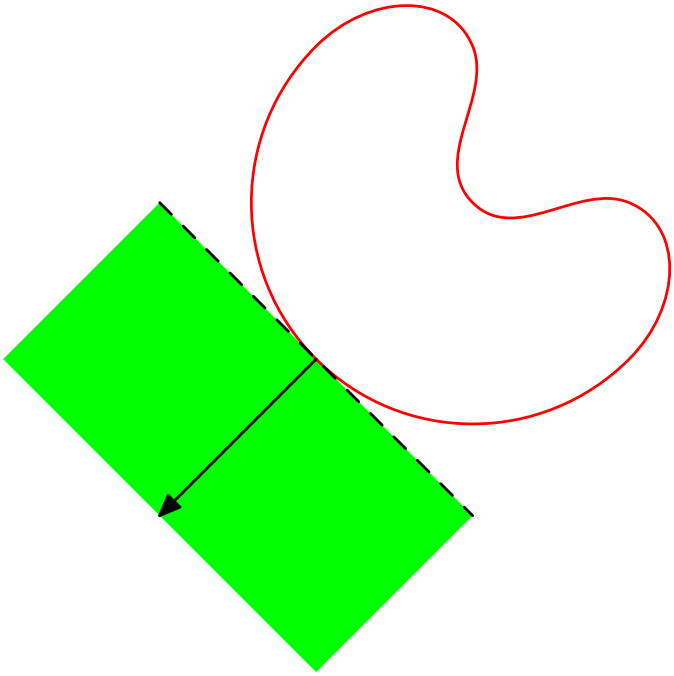
$$(u - u_0) \cdot \hat{n} = 0 \tag{4.18}$$

We can encode this equation as a COCO-compatible function `genplan`:

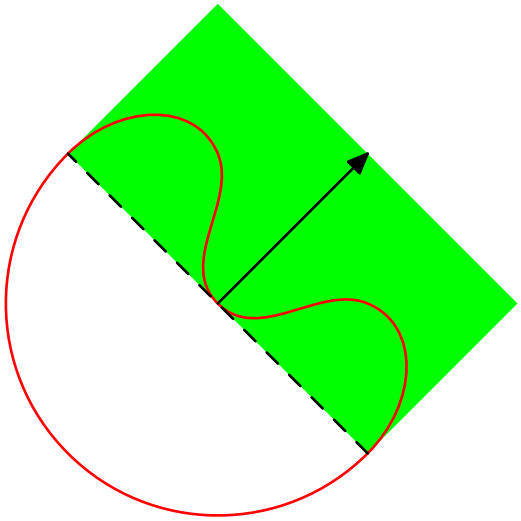
Listing 4.21: A plane passing through u_0 with normal V

```
function [data y] = genplan(prob, data, u)

  y = (u'-data.u0)*data.V';

end
```

Here, `data.u0` is a $1 \times n$ array that represents $u_0 \in \mathbb{R}^n$ and `data.V` is also a $1 \times n$ array that refers to an outward-facing normal of the plane. Notably, the integer n is not part of the encoding, and the function is therefore independent of the problem dimension.



(a) Domain decomposition with a locally convex boundary.



(b) Domain decomposition with a locally concave boundary.

Figure 4.2: Domain decomposition: the red curve is the boundary of the atlas. The arrow points in the direction of the outward normal and the dashed line defines the boundary between halfspaces. The green region indicates the halfspace that the worker will attempt to cover. While a locally convex boundary results in a good split, a locally concave boundary will lead to a split that might contain some regions of the atlas.

A planar constraint can thus be added to the problem by invoking the `coco_add_func` function. For example:

```
data.u0 = [0 0 1];
data.V = [0 0 1];
prob = coco_add_func(prob, 'genplan1', @genplan, ...
         data, 'inactive', 'p1', 'uidx', (1:numel(data.u0)));
```

Here, the planar constraint being added is the $z = 1$ plane. We can now restrict the solution manifold of this problem to only lie in the half-space $z < 1$ as follows:

```
coco(prob, 'run_init', [], 2,...
  {'x' 'y' 'z' 'p1'},{[] [] [] [-Inf 0]});
```

here, `'x' 'y' 'z'` are string labels for continuation parameters that have been added earlier in the problem construction. The call restricts `p1` to take only negative values, which corresponds to all points where:

$$(u - u_0) \cdot \hat{n} < 0 \tag{4.19}$$

i.e the half-space $z < 1$.

Thus, we now have an easy way to add the constraint that was just discussed. We add a vertex of the boundary chart as `data.u0`, and the outward facing direction as `data.V`, and then add this to the continuation problem structure using the `coco_add_func` utility.

We can use the method just described to repeatedly add constraints to the continuation problem structure. It is up to the user to specify constraints that evaluate to a nonzero space (e.g. the user *can* specify both $z > 1$ and $z < 1$ to the same continuation problem; however, in this case the solution manifold is just the empty set). We make use of this facility by constructing a function `restrict_prob` that adds a list of planar restrictions to the continuation problem structure as follows:

Listing 4.22: Function that adds a list of planar restrictions

```
function [prob ps] = restrict_prob(prob, u, V)

for i=1:numel(u)
if ~isempty(u{i})
  data.u0 = u{i}';
  data.V = V{i}';
  prob = coco_add_func(prob, strcat('genplan', num2str(i)),...
           @genplan, data, 'inactive', ...
```

```
            strcat('p', num2str(i)), 'uidx',(1:numel(u0)));
  ps{end+1} = strcat('p', num2str(i));
end
end

end
```

Here, the input variables are the continuation problem structure `prob`, a cell array `u` of points on the respective planes and a cell array `v` of the normals to the respective planes. The `ps` output argument contains the string labels assigned to each planar restriction added to `prob`. Thus, the output `prob` will be restricted to a computational domain which is the intersection of all the half-spaces specified by `u` and `v`.

Thus, to inform a worker of its domain of operation, the cell arrays `u` and `v` are given to the worker which then uses the `restrict_prob` function to restrict the continuation to the corresponding computational domain.

The complete code listing where the master specifies a domain and the worker adds it is specified in appendix A.2. It is evident that this method is only a heuristic, and is not guaranteed to be optimal. There are certain domains for which it gives good results: namely, convex domains (A *convex* domain is one where for every pair of points in the domain, the line that connects them also lies in the domain). In fact, this idea for domain decomposition was inspired by the definition of a convex domain: boundary points on convex domains are such that the method that we just described gives good domain decompositions.

4.4 Experimental Results

We have tested our method successfully when computing manifolds; we show some results in this section.

First, we consider the computation of a cylindrical manifold with symmetry axis parallel to the u_3 axis and running through the point (1,0,0). The serial code for obtaining this is given below

```
prob = coco_add_func(coco_prob(), 'cylinder', @cylinder, [], ...
  'zero', 'u0', [1;0;0]+sqrt([0.5;0.55;0]) );
prob = coco_add_pars(prob, '', [1, 2, 3], {'x' 'y' 'z'});
prob = coco_set(prob, 'cont', 'h', 0.4, 'almax', 20, 'PtMX', 20);
coco(prob, 'runname', [], 2, {'x' 'y' 'z'}, {[] [] [-100 100]});
```

Charts	Serial(in sec)	Parallel(in sec)
20	0.743509	1.8148
200	9.267634	7.9223
2000	162.970819	62.7388

Table 4.1: Comparison between serial and parallel computation of cylinder

Charts	Merging($x = 1$)(in sec)	Merging ($z = 0$)(in sec)
20	0.087804	0.112773
200	0.872065	2.022711
2000	39.113174	2.186090

Table 4.2: Comparison of merge times across different boundaries

and the corresponding COCO-compatible encoding is given below:

Listing 4.23: coco-compatible encoding of cylinder

```
function [data y] = cylinder(prob, data, u)
  y = (u(1)-1)^2 + u(2)^2 - 1;
end
```

The results are given in table 4.1. We see a dramatic improvement in performance only when the number of charts to be computed is large. Concomitantly, when the domain is split across $z = 1$, the time to merge remains the same after sufficient number of charts have been computed; this is easy to see as the boundary between the two cylinders has the same size, so we would expect the merge time to be same. In the case when the domain is split across the $x = 1$ plane, we see that merge time does increase with the size of the manifold, as the boundary between the two computed manifolds increases (see table 4.2).

In the previous experiment, we had to specify the computational domains using features provided by the COCO toolbox. Next, we experimented with using only the automatic domain division that we discussed in the previous section. One thing that we observed was that frequently, the final domains had numerous gaps in the cover. After some investigation, we realized that the gaps were due to the nature of the merge: since the newly computed atlas is separated from the growing manifold, the charts closest to the manifold will be the boundary charts of the atlas. These boundary charts, by their very nature, have a preponderance of gaps. These gaps are included in the

Charts	Serial(in sec)	Parallel(in sec)
100	9.735586	8.804268
200	12.197312	7.139654
500	27.452409	14.245870

Table 4.3: Comparison of manifold computation time. The parallel algorithm was run with two parallel processes

global atlas after merging. We can think of several methods by which to avoid this situation: we may change the atlas algorithm employed by the workers so that such gaps are also covered. We may select a plane such that it overlaps significantly with the global manifold; hence, the same region would be covered twice, this removes the possibility of gaps with little overhead.

Timing results are given in table 4.3 for the case of covering the plane $z = 1$.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, we considered the problem of determining the implicitly defined solution manifold given by the roots of nonlinear system of equations. We discussed the theory of continuation algorithms and how they can be used to determine the solution manifolds of such systems. We then discussed the continuation toolbox COCO and illustrated how it can be used to solve an example problem. We took a brief look into the design of the COCO framework; how atlas algorithms can be designed as finite state machines. Then, we considered the different atlas algorithms that were described in [9]. We saw that, for higher dimensional manifolds, the computation of an atlas could be potentially made faster with the help of parallellism. We then showed how this was possible in the case of 1D manifolds; and then extended our ideas to implement such a design for 2D manifolds.

We have proposed a novel method for increasing the speed of manifold covering: by offsetting the process of manifold covering to an array of worker processes, we achieve speedup in the process of manifold covering. Since manifold covering operates at a lower level than problem definition, this speedup is available to a lot of problems, automatically. The speedup was obtained with minimal modifications to the existing codebase; in fact, it is implemented as a separate subclass of the COCO core class, `AtlasBase`. We discovered that it was better to design such a class rather than to change the core of the COCO toolbox.

We have shown how the manifold covering works for simple manifolds; the next step would be applying these methods to more complex manifolds. During development, we used up to two worker processes; future work would explore the effect of having an order of magnitude larger number of processes. In particular, the falling cost per processor and the growing density of processors on a single chip mean that future work would focus on using hundreds of processes for computing large manifolds; in that case, the

process of merging by the master would most likely become a bottleneck. Techniques would have to be developed to perhaps divide the manifold better amongst multitudes of processes; or maybe the overall merge process itself could be distributed, allowing some redundant computation in exchange for faster overall computation time.

We have described only one way to achieve parallelism; there are many other strategies that can be attempted. In particular, most CPU chipsets come with an onboard or dedicated GPU; many of the machines in the TOP 500 are equipped with both CPU's and GPU's. To extract all this computational power; we might incorporate the usage of GPU's to solve specific subproblems; like a Newton method implemented on the GPU. This would allow one to use a combined strategy: have heavy MATLAB processes determined the overall division of the solution domain; but each process could access the GPU to speedup its computation. We see that there are many issues of concurrent access to the GPU that the user must handle before this method can be successfully used.

Although we discussed various kinds of computational geometric data structures, in our current work, we have not used any special structure to store geometrical information; instead we relied on COCO's functionality for specifying and restricting the domain of computation to limit the computation of manifold to designated regions. As we discussed in the previous paragraph, when the number of workers is large enough, the bottleneck in the computation becomes the merge process. Any improvement in covering time will hence have to be from a corresponding increase in the speed of merging.

We immediately see another fruitful area of research: specifying the domain decomposition is usually a very complex task, and only the simplest characterizations of domain decompositions are in common use (mostly using planes and hyperplanes). However, COCO has no such restriction; it makes available to us the use not only of curved domain boundaries, but such boundaries that can be defined implicitly; thus all the mathematical functions for specifying manifolds are now available to us to divide the domain in interesting nontrivial ways. We believe this is perhaps the most exciting area of further research; ideas from COCO might be incorporated into more general tools for parallel computation.

In our work, we simply chose a boundary point for further continuation;

there are better ways to select a base point, perhaps based on the tangent space properties of the boundary points. This might allow for a better manifold construction when the domain has high curvature. In particular, we want such areas to be covered by *smaller* charts so that interesting parts of the domain are not skipped.

We explored parallel manifold covering in 2D; naturally, the next step is to evaluate these methods in higher dimensions. We are excited by the possibilities that this presents for more research into this area. In particular, we believe that ideas developed for tackling problems in this domain might be applicable to other fields as well.

APPENDIX A

MATLAB CODES

Here, we shall list the complete code of most of the experiments described in the thesis.

A.1 Parallel 1D Atlas Algorithm: Expanding Boundary Method

Listing A.1: atlas_1d_min.m

```
classdef atlas_1d_min < AtlasBase

  properties (Access=private)
    boundary = {};
    cont     = struct();
    flagarray   = [];        % flagarray used in parallel comp
    next_pt     =  0 ;
  end

  methods (Access=private)
    function atlas = atlas_1d_min(prob, cont, dim)
      assert(dim==1, '%s: wrong manifold dimension', mfilename);
      atlas      = atlas@AtlasBase(prob);
      atlas.cont = atlas.get_settings(cont);
    end
  end

  methods (Static)
    function [prob cont atlas] = create(prob, cont, dim)
      atlas = atlas_1d_min(prob, cont, dim);
      prob  = CurveSegment.add_prcond(prob, dim);
    end
  end

  methods (Static, Access=private)
    cont = get_settings(cont)
  end

  methods (Access=public)
    [prob atlas cseg correct] = init_prcond(atlas, prob, chart)
```

```
    [prob atlas cseg flush]   = init_atlas (atlas, prob, cseg)
    [prob atlas cseg]         = flush      (atlas, prob, cseg)
    [prob atlas cseg correct] = predict    (atlas, prob, cseg)
    [prob atlas cseg flush]   = add_chart  (atlas, prob, cseg)
  end

  methods (Access=private)
    flag = isneighbor(atlas, chart1, chart2)
    [atlas cseg] = merge(atlas, cseg)
  end

end
```

Listing A.2: predict.m

```
function [prob atlas cseg correct] = predict(atlas, prob, cseg)

if labindex > 1  % logic for worker process
  [chart xp s h] = atlas.boundary{1,:};
  prcond = struct('x', chart.x, 'TS', chart.TS, 's', s, 'h', h);
  th             = atlas.cont.theta;
  if th>=0.5 && th<=1
    xp           = chart.x+(th*h)*(chart.TS*s);
    [prob cseg] = CurveSegment.create(prob, chart, prcond, xp);
    [prob ch2]  = cseg.update_TS(prob, cseg.curr_chart);
    h           = h*(ch2.TS'*chart.TS);
    xp          = chart.x+h*(ch2.TS*s);
    prcond      = ...
      struct('x', chart.x, 'TS', ch2.TS, 's', s, 'h', h);
  end
  [prob cseg] = CurveSegment.create(prob, chart, prcond, xp);
  correct     = true;
else % logic for master process
  [prob cseg]   = ...
    CurveSegment.create_initial(prob, atlas.boundary{1,1}, false);
  correct     = false;

end

end
```

Listing A.3: init_prcond.m

```
function [prob atlas cseg correct] = ...
  init_prcond(atlas, prob, chart)

chart.R       = 0;
chart.pt      = -1;
chart.pt_type = 'IP';
chart.ep_flag = 1;
[prob cseg]   = CurveSegment.create_initial(prob, chart);
```

```
if labindex == 1 % logic for master process
    correct       = cseg.correct;
else % logic for worker process
    correct = false;
end

end
```

Listing A.4: add_chart.m

```
function [prob atlas cseg flush] = ...
  add_chart(atlas, prob, cseg)

if labindex > 1 % logic for worker process

  chart    = cseg.curr_chart;
  chart.pt = chart.pt+1;
  if chart.pt>=atlas.cont.PtMX
    chart.pt_type = 'EP';
    chart.ep_flag = 1;
  end
  [prob cseg] = cseg.add_chart(prob, chart);
  flush       = true;

  if ~atlas.isneighbor(cseg.ptlist{1}, cseg.ptlist{end})
    cseg.ptlist{end}.pt_type = 'GAP';
    cseg.ptlist{end}.ep_flag = 2;
    cseg.Status     = cseg.CurveSegmentCorrupted;
  end

else % logic for master process

  cseg.Status = ~CurveSegmentBase.CurveSegmentOK;
  cseg.isInitialSegment = false;
  flush       = true;

end

end
```

Listing A.5: flush.m

```
function [prob atlas cseg] = flush(atlas, prob, cseg)

if labindex == 1
  if cseg.isInitialSegment
    atlas.flagarray = ones(1, numlabs);
  else

    [cseg, src, tag] = labReceive();
```

```
    atlas.flagarray(src) = 1;
    chart    = cseg.curr_chart;
    atlas.next_pt = atlas.next_pt+1;
    chart.pt = atlas.next_pt+1;
    if chart.pt>=atlas.cont.PtMX
      chart.pt_type = 'EP';
      chart.ep_flag = 1;
    end

  end

  if cseg.Status==cseg.CurveSegmentOK
    [atlas cseg] = atlas.merge(cseg);
  end
  [prob atlas cseg] = atlas.flush@AtlasBase(prob, cseg);
  if cseg.Status==cseg.CurveSegmentOK
    if isempty(atlas.boundary) || ...
      (atlas.boundary{1,1}.pt>=atlas.cont.PtMX)
      cseg.Status = cseg.BoundaryPoint;
    end
  end
  % logic for scheduling job amongst workers
  [atlas.boundary, atlas.flagarray] = ...
    scheduleSend(atlas.boundary, ...
      cseg.Status, atlas.flagarray);
else

  if ~cseg.isInitialSegment
    labSend(cseg,1);
  end

  dat = labReceive(1);
  cseg.Status = dat{1}; % stop FSM
  atlas.boundary = dat{2};

end

end

function [boundary, farray] = ...
  scheduleSend(boundary, status, farray)

if  status == 0
  for i = 2:numlabs
    for j = 1:size(boundary, 1)
      if farray(i) & boundary{j,end}
        farray(i) = 0;
        boundary{j,end} = 0;
        labSend({status boundary(j, :)}, i);
        break
      end
    end
  end

else % terminate all labs
```

```
  pause(3);
  for i = 2:numlabs
    if ~farray(i)
      farray(i) = 1;
      dat = labReceive(i);
    end
    labSend({status []}, i);
  end

end

end
```

Listing A.6: get_settings.m

```
function cont = get_settings(cont)

defaults.h     = 0.1;
defaults.PtMX  = 50;
defaults.theta = 0.5;
defaults.almax = 10;
defaults.Rmarg = 0.95;
cont           = coco_merge(defaults, cont);
cont.almax     = cont.almax*pi/180;

end
```

Listing A.7: init_atlas.m

```
function [prob atlas cseg flush] = init_atlas(atlas, prob, cseg)

chart           = cseg.curr_chart;
chart.pt        = 0;
chart.R         = atlas.cont.h;
chart.s         = [1, -1]*sign(atlas.cont.PtMX);
atlas.cont.PtMX = abs(atlas.cont.PtMX);
chart.pt_type   = 'EP';
chart.ep_flag   = 1;
[prob cseg]     = cseg.add_chart(prob, chart);
flush           = true;

end
```

Listing A.8: isneighbor.m

```
function flag = isneighbor(atlas, chart1, chart2)

al  = atlas.cont.almax;
ta  = tan(al);
```

```
R   = atlas.cont.h;
x1  = chart1.x;
x2  = chart2.x;
dx  = x2-x1;
x1s = chart1.TS*(chart1.TS'*dx);
x2s = chart2.TS*(chart2.TS'*dx);
dst = [norm(x1s), norm(x2s), norm(dx-x1s), norm(dx-x2s), ...
       subspace(chart1.TS, chart2.TS)];
dstmx = [R, R, ta*norm(x1s), ta*norm(x2s), al];
flag  = all(dst<dstmx);

end
```

Listing A.9: merge.m

```
function [atlas cseg] = merge(atlas, cseg)

chart = cseg.ptlist{end};
R     = atlas.cont.h;
h     = atlas.cont.Rmarg*R;
nb    = cell(2,5);
for k=1:2
  sk      = chart.s(k);
  xk      = chart.x+h*(chart.TS*sk);
  nb(k,:) = {chart, xk, sk, h, 1};
end
for i=size(atlas.boundary,1):-1:1
  chart2 = atlas.boundary{i,1};
  if atlas.isneighbor(chart, chart2)
    x2 = atlas.boundary{i,2};
    if norm(chart.TS'*(x2-chart.x))<R
      atlas.boundary(i,:) = [];
    end
    for k=size(nb,1):-1:1
      x1 = nb{k,2};
      if norm(chart2.TS'*(x1-chart2.x))<R
        nb(k,:) = [];
      end
    end
  end
end
atlas.boundary = [nb; atlas.boundary];
if isempty(atlas.boundary)
  chart.pt_type    = 'EP';
  chart.ep_flag    = 1;
  cseg.ptlist{end} = chart;
end

end
```

A.2 Parallel Domain Decompostion and Merging

In this section, we present the code that handles the overall domain decomposition and merging.

Listing A.10: 'Construct and manage parallel covering'

```
function [atlas] = demo()
% the atlas_merge subdirectory must
% contain the following functions:
% merge_recursive.m
% substract_halfspace.m
% atlas_close.m
% atlas_merge.m
% isclose.m
% get_settings.m

addpath('./atlas_merge');

% cleanup before starting new parallel comp

stat = matlabpool('size');
if stat > 0
    matlabpool close;
end

% get settings
cont = get_settings;

matlabpool open local 3

% Master: Should have all the initial problem
% parameters, create the prob
% structure, divide the domain and give the
% parameters to the workers.

% Worker: do the computation, return the
% finished atlas back to master
spmd
  if labindex == 1
    master_code();
  else
    worker_code(labindex);
  end
end
matlabpool close;

% write separate functions to get around
% transparency violation requirements of
% parallel computing toolbox

function master_code()
```

```
% code for master

cont = get_settings;
cont.ptmx = 100;
rprefix = 'run_of_lab_';
u0 = [0;0;0];
ptcnt = 0;

% pdata holds the domain restrictions

pdata.u  = cell(1,numlabs-1);
pdata.V  = cell(1,numlabs-1);
pdata.ct = cell(1,numlabs-1);

% create the problem structure

prob  = create_prob(u0, {}, {});
prob = coco_add_func(coco_prob(), 'cylinder', ...
  @cylinder, [], 'zero', 'u0', u0);
% store the minimum dimension in cont
cont.mind = prob.efunc.x_dim;

prob = coco_add_pars(prob, '', [1, 2, 3], ...
  {'x' 'y' 'z'});
prob = coco_set(prob, 'cont', 'h', 0.4, ...
  'almax', 5, 'PtMX', 1);

% run the coco algorithm, get an initial atlas
% CAUTION: if the algo terminates before
% covering even this small number of
% charts, it indicates premature termination
% and MUST EXIT

coco(prob, 'run_init', [], 2, ...
  {'x' 'y' 'z' 'p0'},{[] [] [] []});
atlas_arr = {coco_bd_read('run_init')};

% logic to manage the workers
% if 1, worker is available; if 0 its busy

workers = ones(1, numlabs-1);
domcov  = zeros(1, numlabs-1);

% 'domain_covered' becomes true when there are
% no remaining continuation
% directions
% 'max_reached' is true when the maximum number
% of charts is computed
% continuation stops either because:
% a. The whole domain has been covered
% b. max number of charts has been reached

% function to pick a minimal set of dimensions

f = @(x) x([1:cont.mind]);
atlas1 = [];
```

```
count = 1;

while numel(atlas_arr{1}.charts) -1 < cont.ptmx

  % divide the domain, keep track of it
  % send domain to workers; keep track of the
  % workers that are busy and which
  % are free

  % pick a boundary chart, and pick its boundary point

  bdchart = [];
  anatlas = atlas_arr{1};

  % We're looking for a boundary direction that's
  % not been used yet
  % NOTE: if the bv values of the chart are empty,
  % the following loop will NOT remove the chart id from
  % the boundary array

  rindx = 1;
  while ~isempty(anatlas.boundary)
    if ~isempty(anatlas.charts{anatlas.boundary(1)}.bv)

      bdchart = anatlas.charts{anatlas.boundary(1)};
      rindx = randi(numel(anatlas.charts{anatlas.boundary(1)}.bv));
      anatlas.charts{anatlas.boundary(1)}.bv(rindx) = [];

      if isempty(anatlas.charts{anatlas.boundary(1)}.bv)
        anatlas.boundary(1) = [];
      end

      break;

    % if for any reason the bv's are already
    % empty, remove this
    % from boundary

    else

    anatlas.boundary(1) = [];

    end

    if isempty(anatlas.boundary)
      disp('All boundary points used.')
    end

  end

  atlas_arr{1} = anatlas;


  % set the constraints for the domain. The
  % domain restrictions
  % are planar in nature
```

```
if ~isempty(bdchart)
  indworkr = find(workers, 1);
  if ~isempty(indworkr)
    pdata.u{indworkr} = ...

  f(bdchart.x+min(bdchart.R, ...
    bdchart.v(bdchart.bv(rindx)))* ...
    (bdchart.TS*bdchart.s(:,bdchart.bv(rindx))));

  pdata.V{indworkr} = ...
    f((bdchart.TS*bdchart.s(:,bdchart.bv(rindx)))/ ...
      norm((bdchart.TS*bdchart.s(:,bdchart.bv(rindx)))));

  pdata.ct{indworkr} = [0 Inf];

  % send the data to the workers

  labSend({50, pdata, 1}, indworkr +1);
  workers(indworkr) = 0;

  % the rest of the workers get this kind of restriction

  pdata.ct{indworkr} = [-Inf 0];
  end
end

% is there another worker trying to send data to master?
[isDataAvail,srcWkrIdx,tag] = labProbe;
if (isDataAvail)

  ret = labReceive(srcWkrIdx)
  workers(ret{1}) = 1;
  domcov(ret{1}) = ret{2};
  atlas1 = read_atlas(strcat(rprefix, num2str(ret{1}+1)));

end

% get the computed atlases
% store the worker-computed atlas in a global cell array

if ~isempty(atlas1) && ~isempty(atlas1.charts)

  atlas_arr{end+1} = atlas1;
  ptcnt = ptcnt + numel(atlas_arr{end}.charts)-1;
  atlas1 = [];

end

% merge manifold into single atlas array
% it is here that further changes might be made
% so that master merges
% more than one atlas at a time

if numel(atlas_arr) > 1
```

```
    atlas_arr{1} = ...
      atlas_merge(atlas_arr{1}, atlas_arr{2}, cont);
    atlas_arr(2) = [];

  end

  % are there possibly any more continuation directions?

  if isempty(atlas_arr{1}.boundary)
    break
  end

  % Check if all workers are busy, if so, then wait.

  if ~any(workers)
    ret = labReceive();
    workers(ret{1}) = 1;
    domcov(ret{1})  =  ret{2};
    atlas1 = ...
      coco_bd_read(strcat(rprefix, num2str(ret{1}+1)));
  end

end

% tell all workers to quit

for i=1:numel(workers)
  if workers(i)
    labSend({[],[],0}, i+1);
  else
    labReceive(i+1);
    labSend({[],[],0}, i+1);
  end
end

end

function worker_code(labindex)

% code for worker

while(true)
  rprefix = 'run_of_lab_';
  runname = strcat(rprefix, num2str(labindex));

  % create the problem structure
  % get data from master

  temp = labReceive(1);
  if ~temp{3}
  break;
  end

  % create problem data structure using
  % restrictions given by master
```

```
  pdata = temp{2};

  % the initial point is the initial point of the
  % last added constraint

  [prob ps] = create_prob(pdata.u{labindex-1}, ...
  pdata.u, pdata.V);

  % set some settings

  prob = coco_set(prob, 'cont',  'PtMX', temp{1});

  % run the computation

  prob = coco(prob, runname, [], 2, {'x' 'y' 'z' ps{:}}, ...
    {[] [] [] pdata.ct{find(~cellfun('isempty', pdata.ct))}});

  % did the worker complete because it covered
  % the domain or because it
  % exceeded the maximum number of points?

  atlas1 = coco_bd_read(runname);
  flag = numel(atlas1.charts) < temp{1} + 1;

  % store/send the data

  labSend({labindex-1 flag},1);
end

end
```

The functions used by `atlas_merge` are described in Section 4.3.3, and are given here for completeness. They must all be in the MATLAB path; preferably, in the same directory as the `atlas_merge` function.

Listing A.11: `atlas_merge`: Merge two atlases

```
function atlas = atlas_merge(atlas1, atlas2, cont)

% merge 2 atlas, atlas1 and atlas2, store result in
% atlas

% the atlas with greater number of charts is
% assigned to atlas

if atlas1.charts{end}.id>=atlas2.charts{end}.id
  atlas = atlas1;
else
  atlas = atlas2;
  atlas2 = atlas1;
end

size = atlas.charts{end}.id;
```

```
% store pairs of coincident charts in a cell
% array

atlas.coincident = {};

neighbors = atlas2.boundary;
bd_charts = atlas.boundary;

% function that merges charts of atlas and atlas2
% charts in both atlases are changed, but no charts
% are moved from one atlas to another

[atlas, atlas2] = ...
  merge_into(size, neighbors, bd_charts, atlas, atlas2, cont);

% for all charts of atlas2 that were not changed
% by the merge_into function, update the id's of both
% the charts and the ordinal numbers stored in their
% nb fields(as they refer to charts whose id has been
% changed)

for i=1:numel(atlas2.charts)
  if ~isfield(atlas2.charts{i},'check')
    atlas2.charts{i}.id = atlas2.charts{i}.id+size;
    atlas2.charts{i}.nb(atlas2.charts{i}.nb>0) = ...
      atlas2.charts{i}.nb(atlas2.charts{i}.nb>0)+size;
  end
end

% add the charts from atlas2 into atlas

atlas.charts   = [atlas.charts, atlas2.charts];

% merge the boundary array

atlas.boundary =
  union(atlas.boundary, atlas2.boundary+size);

% update the boundary array so that only charts with
% nonempty bv are referenced by the boundary array

bd_charts      = atlas.charts(atlas.boundary);
idx            = cellfun(@(x) ~isempty(x.bv), bd_charts);
atlas.boundary = atlas.boundary(idx);

% now handle any charts that are coincident

if ~isempty(atlas.coincident)

  for i=1:numel(atlas.coincident)
    pair = atlas.coincident{i};
    nbhd = atlas.charts{pair(1)}.nb
    for j=1:numel(nbhd);
      if nbhd(j) == 0
        continue
```

```
      end
      atlas.charts{nbhd(j)}.nb(atlas.charts{nbhd(j)}.nb ...
        == pair(1)) = pair(2);
    end

    % remove one chart of the pair from the atlas

    atlas.charts{pair(1)} = [];
    atlas.boundary = setdiff(atlas.boundary, pair(1));
    atlas = close_atlas(atlas, cont, pair(2));
  end
end
atlas = rmfield(atlas, 'coincident');

% function for double-recursive merge of two atlases
function [atlas, atlas2] = ...
  merge_into(size, neighbors, ...
    bd_charts, atlas, atlas2, cont)

% input arguments
% atlas:     an atlas
% atlas2:    another atlas
% size:      id of the last chart of atlas
% cont:      structure with predefined settings
% neighbors: boundary charts of atlas2
% bd_charts: boundary charts of atlas

  for i=1:numel(neighbors)
    chart        = atlas2.charts{neighbors(i)};

    % update the id field of the chart if not
    % done so already

    if ~isfield(chart,'check')
      chart.id = chart.id+size;
      chart.nb(chart.nb>0) = chart.nb(chart.nb>0)+size;
      chart.check = [];
    end

    % find all charts in atlas2 that are close to the
    % boundary charts of atlas and have not already
    % been modified

    nbfunc       = @(x) isclose(chart, x, cont);
    idx          = cellfun(nbfunc, atlas.charts(bd_charts));
    close_charts = setdiff(bd_charts(idx), chart.check);
    if ~isempty(close_charts)
      checked    = [0, chart.id];

      % merge one chart from close_chart with the
      % charts in atlas2

      while ~isempty(close_charts)
        [atlas, chart, checked] = ...
            merge_recursive(size, atlas, ...
              chart, close_charts(1), checked, cont);
```

```
        close_charts = setdiff(close_charts, checked);
      end
      chart.check = [chart.check, checked];
      atlas2.charts{neighbors(i)} = chart;

      % recursively select all charts in atlas and merge
      % until no chart in atlas2 is close
      % and not checked

      [atlas, atlas2] = ...
        merge_into(size, chart.nb(chart.nb>size)-size, ...
          bd_charts, atlas, atlas2, cont);
    end
  end
end

end
```

Listing A.12: coco-compatible encoding of general plane function

```
function [data y] = genplan(prob, data, u)

% equation of an n dimensional plane, given
% u0     A point on the plane 1 X n
% V      Normal to the plane  1 X n

  y = (u'-data.u0)*data.V';

end
```

Listing A.13: Create the problem structure and apply constraints

```
function [prob] = create_prob(u0, u, V)

% create and return a problem structure. The initial point is
% hardcoded, while the constraints are specified as a cell
% array in `u` and `V` corresponding to the equation of a
% plane as specified in  genplan

  % specify original problem here

  prob = coco_add_func(coco_prob(), 'cylinder', ...
    @cylinder, [], 'zero', 'u0', u0);
  prob = coco_add_pars(prob, '', [1, 2, 3], ...
    {'x' 'y' 'z'});
  prob = coco_set(prob, 'cont', 'h', 0.4, 'almax', 20, ...
    'PtMX', 10);

  % add domain restrictions corresponding to the initial point.
  % Note: the initial point must be one that satisfies ALL these
  % restrictions! That is guaranteed by the fact that we
```

```
  % choose the initial point from the solution manifold

  for i=1:numel(u)
    data.u0 = u{i}';
    data.V = V{i}';
    prob = coco_add_func(prob, strcat('genplan', num2str(i)), ...
      @genplan, data, 'inactive', strcat('p', num2str(i)), ...
        'uidx',(1:numel(u0)));
  end
end
```

```
function flag = iscoincident(chart1, chart2)

x1   = chart1.x;
x2   = chart2.x;
dx   = x2-x1;
phi1 = chart1.TS'*dx;
phi2 = chart2.TS'*dx;
x1s  = chart1.TS*(phi1);
x2s  = chart2.TS*(phi2);
dst  = [norm(x1s), norm(x2s), norm(dx-x1s), norm(dx-x2s), ...
        subspace(chart1.TS, chart2.TS)];
dstmx = [1e-10, 1e-10, 1e-10, 1e-10, 1e-10];
flag  = false;
if all(dst<dstmx)
    flag = true;
end

end
```

```
function [atlas, chart1, checked] = ...
    merge_recursive(size, atlas, chart1, k, checked, cont)
checked(end+1) = k;
chartk = atlas.charts{k};
if isclose(chart1, chartk, cont)
  dx     = chartk.x-chart1.x;
  phi1   = chart1.TS'*dx;
  phik   = chartk.TS'*(-dx);
  test1  = chart1.v.*(chart1.s'*phi1)-norm(phi1)^2/2;
  testk  = chartk.v.*(chartk.s'*phik)-norm(phik)^2/2;
  flag1  = (test1>1e-10);
  flagk  = (testk>1e-10);
  if ~isempty(find(flag1, 1))
  chart1 = ...
    subtract_half_space(chart1, test1, ...
      phi1, flag1, chartk.id, cont);
  end
  if ~isempty(find(flagk, 1))
    chartk = ...
      subtract_half_space(chartk, testk, ...
        phik, flagk, chart1.id, cont);
  end
  atlas.charts{k} = chartk;
  check = setdiff(chartk.nb(chartk.nb<=size), checked);
```

```
  while ~isempty(check)
      [atlas, chart1, checked] = ...
          merge_recursive(size, atlas, ...
            chart1, check(1), checked, cont);
      check = setdiff(chartk.nb(chartk.nb<=size), checked);
  end
elseif iscoincident(chart1, chartk)
  atlas.coincident = [atlas.coincident, [chart1.id, k]];
end

end
```

```
function atlas = close_atlas(atlas, cont, k)

chart        = atlas.charts{k};
chart.v      = chart.R*ones(numel(chart.v),1);
close_charts = chart.nb;
checked      = [0, chart.id];
while ~isempty(close_charts)
[atlas, chart, checked] = ...
    merge_recursive(atlas.charts{end}.id, atlas, ...
      chart, close_charts(1), checked, cont);
close_charts = setdiff(close_charts, checked);
end
atlas.charts{k} = chart;
if isempty(chart.bv)
    atlas.boundary = setdiff(atlas.boundary, 1);
end

end
```

```
function flag = isclose(chart1, chart2, cont)

% k contains the minimum dimension of the base points
k    = cont.mind;
al   = cont.almax;
R    = cont.h;
ta   = tan(al);
t2a  = tan(2*al);
x1   = chart1.x(1:k);
x2   = chart2.x(1:k);
dx   = x2-x1;
phi1 = chart1.TS(1:k,:)'*dx;
phi2 = chart2.TS(1:k,:)'*dx;
x1s  = chart1.TS(1:k,:)*(phi1);
x2s  = chart2.TS(1:k,:)*(phi2);
dst  = [norm(x1s), norm(x2s), norm(dx-x1s), norm(dx-x2s), ...
        subspace(chart1.TS(1:k,:), chart2.TS(1:k,:))];
n1mx  = ta*min(R,norm(x1s))+t2a*max(0,norm(x1s)-R);
n2mx  = ta*min(R,norm(x2s))+t2a*max(0,norm(x2s)-R);
dstmx = [2*R, 2*R, n1mx, n2mx, 2*al];
flag  = false;
if all(dst<dstmx);
  test1 = chart1.v.*(chart1.s'*phi1)-norm(phi1)^2/2;
```

```
  test2 = chart2.v.*(chart2.s'*phi2)+norm(phi2)^2/2;
  flag  = any(test1>0) && any(test2<=1e-10);
end

end
```

Listing A.14: Split a chart using a half space

```
function chart = ...
  subtract_half_space(chart, test, phi, flag, NB, cont)

k        = find(flag & ~circshift(flag, -1), 1);
flag     = circshift(flag, -k(1));
test     = circshift(test, -k(1));
chart.s  = circshift(chart.s, [0, -k(1)]);
chart.v  = circshift(chart.v, -k(1));
chart.nb = circshift(chart.nb, [0, -k(1)]);
j        = find(~flag & circshift(flag, -1), 1);
vx1      = chart.v(j)*chart.s(:,j);
vx2      = chart.v(j+1)*chart.s(:,j+1);
nvx1     = vx1-test(j)/((vx2-vx1)'*phi)*(vx2-vx1);
vx1      = chart.v(end)*chart.s(:,end);
vx2      = chart.v(1)*chart.s(:,1);
nvx2     = vx1-test(end)/((vx2-vx1)'*phi)*(vx2-vx1);
chart.s  = [chart.s(:,1:j), nvx1/norm(nvx1), nvx2/norm(nvx2)];
chart.v  = [chart.v(1:j); norm(nvx1); norm(nvx2)];
chart.nb = [chart.nb(1:j+1), NB];
chart.bv = find(chart.v>cont.Rmarg*chart.R);

end
```

```
function cont = get_settings

cont.h     = 0.2;
cont.theta = 0.5;
cont.almax = 10;
cont.Rmarg = 0.95;
cont.Ndirs = 5;
cont.almax     = cont.almax*pi/180;
cont.ptmx = 100;
end
```

REFERENCES

[1] L. F. Richardson and S. Chapman, *Weather prediction by numerical process*. Dover Publications New York, 1965.

[2] C. R. Chen and T. L. Steiner, "Managerial ownership and agency conflicts: a nonlinear simultaneous equation analysis of managerial ownership, risk taking, debt policy, and dividend policy," *Financial Review*, vol. 34, no. 1, pp. 119–136, 1999.

[3] D. A. Hsieh, "Chaos and nonlinear dynamics: application to financial markets," *The Journal of Finance*, vol. 46, no. 5, pp. 1839–1877, 1991.

[4] L. Grigorev and M. Salikhov, "Financial Crisis 2008," *Problems of Economic Transition*, vol. 51, no. 10, pp. 35–62, 2009.

[5] F. Hoppensteadt, "Predator-prey model," *Scholarpedia*, 2006.

[6] O. Malcai, O. Biham, P. Richmond, and S. Solomon, "Theoretical analysis and simulations of the generalized Lotka-Volterra model," *Physical Review E*, vol. 66, no. 3, p. 031102, 2002.

[7] W. Joubert and S.-Q. Su, "An analysis of computational workloads for the ORNL Jaguar system," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 247–256.

[8] M. E. Henderson, "Higher-dimensional continuation," in *Numerical continuation methods for dynamical systems*. Springer, 2007, pp. 77–115.

[9] H. Dankowicz and F. Schilder, *Recipes for Continuation*. SIAM, 2013.

[10] M. E. Henderson, "Multiparameter parallel search branch switching," *International Journal of Bifurcation and Chaos*, vol. 15, no. 03, pp. 967–974, 2005.

[11] J. R. Munkres, *Analysis on manifolds*. Addison-Wesley Publishing Company, 1991.

[12] D. P. Dobkin, A. R. Wilks, S. V. Levy, and W. P. Thurston, "Contour tracing by piecewise linear approximations," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 4, pp. 389–423, 1990.

[13] M. Henderson, "Continuation methods," 2007, accessed: 2013-12-16. [Online]. Available: http://web.archive.org/web/20130513140731/http://www.research.ibm.com/people/h/henderson/Continuation/ContinuationMethods.html

[14] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, 2008.

[15] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *ACM Siggraph Computer Graphics*, vol. 21. ACM, 1987, pp. 163–169.

[16] W. Schroeder, L. Avila, K. Martin, W. Hoffman, and C. Law, "The Visualization Toolkit-Users Guide," *Kitware, Inc*, 2001.

[17] A. Fossati, J. Gall, H. Grabner, X. Ren, and K. Konolige, *Consumer Depth Cameras for Computer Vision: Research Topics and Applications*. Springer-Verlag New York Incorporated, 2012.

[18] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison et al., "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011, pp. 559–568.

[19] F. Aurenhammer, "Power diagrams: properties, algorithms and applications," *SIAM Journal on Computing*, vol. 16, no. 1, pp. 78–96, 1987.

[20] D. Kirkpatrick, "Optimal search in planar subdivisions," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 28–35, 1983.

[21] E. L. Allgower and S. Gnutzmann, "An algorithm for piecewise linear approximation of implicitly defined two-dimensional surfaces," *SIAM journal on numerical analysis*, vol. 24, no. 2, pp. 452–469, 1987.

[22] F. T. Leighton, *Introduction to parallel algorithms and architectures*. Morgan Kaufmann, 1992.

[23] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[24] B. G. Schuster, "Detection of tropospheric and stratospheric aerosol layers by optical radar lidar," *Journal of Geophysical Research*, vol. 75, no. 15, pp. 3123–3132, 1970.

[25] G. Akinci, M. Ihmsen, N. Akinci, and M. Teschner, "Parallel Surface Reconstruction for Particle-Based Fluids," in *Computer Graphics Forum*. Wiley Online Library, 2012.

[26] W. D. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. The MIT Press, 1999.

[27] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proceeding of International Symposium on Memory Management*, vol. 7, 2007, pp. 103–104.

[28] Mathworks, "MATLAB Parallel Computing Toolbox," 2013, accessed: 2013-12-16. [Online]. Available: http://www.mathworks.com/products/parallel-computing

[29] H. Dankowicz and F. Schilder, "An extended continuation problem for bifurcation analysis in the presence of constraints," *Journal of Computational and Nonlinear Dynamics*, vol. 6, no. 3, 2011.

[30] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[31] W. D. Gropp, "Parallel computing and domain decomposition," in *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations*, 1992, pp. 349–361.

[32] B. Smith, P. Bjorstad, and W. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, 2004.